# TVLA : User's Manual
# (Working Draft)

Roman Manevich* Mooly Sagiv†

December 2, 2007

## Contents



*mailto:rumster@tau.ac.il
†mailto:msagiv@tau.ac.il

# 1 Introduction

This document is intended as a user's manual for the TVLA system. The reader should be familiar with the Three-Valued Logic based Analysis framework described in [**?**] before consulting this manual. The manual is accompanied by an example of the analysis of the reverse function in Figure 1.

The original algorithms in the system were designed and implemented by Tal Lev-Ami [**?**, **?**].

## 1.1 Downloading and installing

The system and latest information is available at:
http://www.cs.tau.ac.il/~tvla/.
Please see the file LICESNE for licensing information.

Installation procedure:

1. TVLA is written in Java and requires J2SE version 1.4.2 (or above), available from http://java.sun.com/j2se/. Before attempting to run TVLA, make sure it is possible to launch Java executables by entering "java" at the command-line prompt.

2. TVLA uses DOT to generate Postscript files and requires Graphviz version 1.12 (or above), available from http://www.research.att.com/sw/tools/graphviz/. After installing Graphviz, make sure the bin sub-directory is added to your path.

3. Extract the archive's content and set the environment variable TVLA_HOME to that location.

4. Add the bin sub-directory to your path (the bin directory contains running scripts for Windows and Linux).

   **IMPORTANT:** Make sure the path does not contain trailing '\' characters or trailing '/' characters.

5. You are now ready to run the system. Enter tvla to see usage information and command-line options.

# 2 Graphical representation

3-valued structures are displayed using graphical representation. For example an input structure for the reverse function is given in Figure 2.

```
/* list.h */
typedef struct node
{
  struct node *n;
  int data;
} *L;
```

(a)

```
/* reverse.c */
#include "list.h"
L reverse(L x) {
    L y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x->n;
        y->n = t;
        t = NULL;
    }
    return y;
}
```

(b)

Figure 1: (a) Declaration of a linked-list data type in C. (b) A C function that uses destructive updating to reverse the list pointed to by parameter x.

%n = {head, tail}
%p = {
sm = {tail:1/2}
n = {head->tail:1/2, tail->tail:1/2}
x = {head}
t[n] = {head->head, head->tail, tail->tail:1/2}
r[n,x] = {head, tail}
}

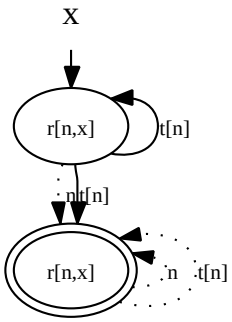


Figure 2: The TVS of an input structure for the reverse function analysis and its graphical representation.

### 2.1 Colors

Colors are used to represent the different values for predicates. Solid black is true (1), dotted black is unknown (1/2), and red is false (0).

### 2.2 Shapes

The values of nullary predicates are displayed in a box titled "nullary". By default, nullary predicates with true values are written inside the box, nullary predicates with false values, are not shown and nullary predicates with indefinite values (1/2) have the value added next to them.

An ellipse represents a node. If the ellipse is double-circled then the node is a summary node, and if it is green then the node is maybe active ($ac = 1/2$). Unary predicates are written within the ellipse. If the value is different from 1 it is appended to predicate's name (i.e., $= 0$ or $= 1/2$).

### 2.3 Edges

Binary predicates are represented as directed arrows between the left and right arguments and annotated by the name of the predicate. If a binary predicate has the same value for both $u_1$->$u_2$ and $u_2$->$u_1$ the two edges are replaced with a bidirectional edge.

## 3 TVP

The specification of the analysis including the control flow graph of the analyzed program is given in a format called TVP (Three Valued Program). A TVP file should end with the extension '.tvp'. The TVP for the analysis of the reverse function is given in Figures 3, 4, and 5. The syntax of a TVP file is given in Figure 6. The syntax is in extended BNF when A $\bowtie$ B denotes a (possibly empty) sequence of A's separated by B's. **Missing displaying different kleene values.**

### 3.1 Predicates

The predicate name can be either <id> or <id>[<id>, ..., <id>]. These are ordinary names. The intention of the <id>[<id>, ..., <id>] format is to denote predicates which are parameterized by source properties such as names of fields and pointer variables. This is especially good for instrumentation predicates. The predicate's arity is determined by the number of variables in parenthesis. For a description of the properties that can be used in predicate declaration see Table 1.

```
// Set of names of program variables.
%s PVar {x, y, t}
/* Program variables definition */
foreach (z in Var) {
        %p z(v1) unique pointer
}
// A predicate to represent the n field of the list data type.
%p n(v1, v2) function
// Is shared instrumentation.
%i is[n](v) = E(v1, v2) (v1 != v2 & n(v1, v) & n(v2, v))
// Reachability instrumentation.
foreach (z in PVar) { %i r[n, z](v) = E(v1) (z(v1) & n*(v1, v)) }
// The t[n] predicate records transitive reflexive reachability between
// list elements along the n field.
%i t[n](v1, v2) = n * (v1, v2) transitive reflexive
// Cyclicity instrumentation.
%i c[n](v) = n+(v, v)
```

Figure 3: The declarations part of the TVP for the reverse function shown in Figure 1.

Instrumentation predicates are declared very similarly to core predicates. They use the same naming mechanism and the same flag specification. The only difference is that for an instrumentation predicate the user has to attach its defining formula. The formula's free variables should match the variables given in the parenthesis (with the exception of precondition free variables explained later).

### 3.2 Formulae

The formula is evaluated in the context of a three valued logical structure using the semantics of Kleene's 3-valued logic. Transitive closure of a general binary formula works as follows, the last pair of variables are the free variables of the subformula, and the first pair of variables are the variables of the resulting TC relation. The formula $\varphi_{cond}?\varphi_{true} : \varphi_{false}$ is an if-then-else formula. If $\varphi_{cond}$ evaluates to true the value of the formula is $\varphi_{true}$. If $\varphi_{cond}$ evaluates to false the value of the formula is $\varphi_{false}$. If $\varphi_{cond}$ is unknown then the result is the join of the values of $\varphi_{true}$ and $\varphi_{false}$, i.e., the value of $\varphi_{true}$ when it is equal to the value of $\varphi_{false}$ and unknown otherwise.

```
/******************************** Generic Actions ****************************/
%action Is_Not_Null_Var(x1) { %t x1 + " != NULL"
    %f { x1(v) } %p E(v) x1(v)
}
%action Is_Null_Var(x1) { %t x1 + " == NULL"
    %f { x1(v) } %p  ! (E(v) x1(v))
}
/******************************** List Actions *******************************/
%action Set_Null_L(x1) { %t x1 + " = NULL"
    { x1(v) = 0 }
    }
    %action Copy_Variable_L(x1, x2) { %t x1 + " = " + x2
        %f { x2(v) }
        { x1(v) = x2(v) }
    }
    %action Get_Next_L(x1, x2) { %t x1 + " = " + x2 + "->" + n
        %f { E(v1) x2(v1) & n(v1, v)}
        { x1(v) = E(v1) x2(v1) & n(v1, v) }
    }
    %action Set_Next_Null_L(x1) { %t x1 + "->" + n + " = NULL"
        %f { x1(v) }
        { n(v1, v2) = n(v1, v2) &  ! x1(v1) }
    }
    %action Set_Next_L(x1, x2) { %t x1 + "->" + n + " = " + x2
        %f { x1(v), x2(v) }
        { n(v1, v2) = n(v1, v2) | x1(v1) & x2(v2) }
    }
```

Figure 4: The actions part of the TVP for the reverse function shown in Figure 1.

```
/* The program's CFG and the effect of its edges */
L1 Set_Null_L(y) L2                 // y = NULL;
L2 Is_Null_Var(x) exit              // while (x != NULL) {
L3 Is_Not_Null_Var(x) L3            // x != NULL
L3 Copy_Variable_L(t, y) L4         // t = y;
L4 Copy_Variable_L(y, x) L5         // y = x;
L5 Get_Next_L(x, x) L6              // x = x->n;
L6 Set_Next_Null_L(y) L7            // y->n = NULL;
L7 Set_Next_L(y, t) L8              // y->n = t;
L8 Set_Null_L(t) L2                 // t = NULL;
                                    // }
exit Assert_ListInvariants(y) error
exit Assert_No_Leak(y) error
```

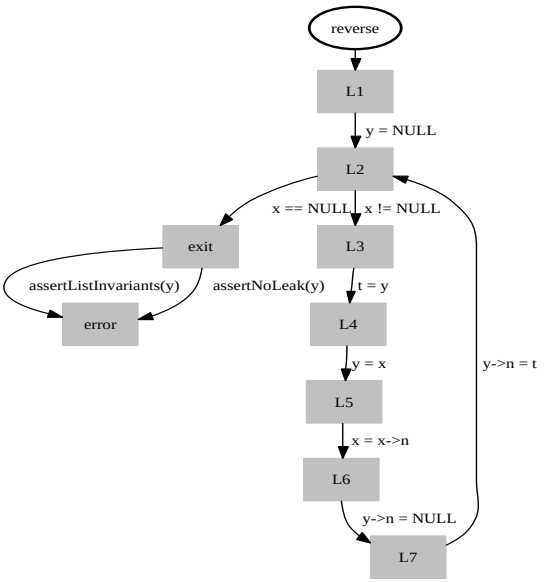


Figure 5: The CFG part of the TVP for the reverse function shown in Figure 1 and its corresponding CFG.

```
<tvp> ::= <decl>* %% <action>* %%                    // TVP file
          <cfg_edge>* [%%<cfg_node>⋈,]
<decl> ::= %s <id> <set_expr>                        // Set declaration
        | %p <pred> ( <var>⋈, ) <prop>*              // Core predicate
        | %i <pred> ( <var>⋈,)                       // Instrumentation predicate
             = <formula> <flags>
        | %r <formula>  ==>  <formula>               // Consistency rule
<pred>::= <id>[ [ <id>⋈, ] ]                         // Predicate name
<kleene> ::= 1 | 0 | 1/2                             // Predicate's flags
<action> ::= %action <id> ( <id>⋈, ) {               // Functional dependencies
          [%t <message>]                             // Atomic values
          [%f { <formula>⋈,]                         // Action declaration
          [%p <formula>]                             // Action title
          (%message <formula> -> <message>)*         // Focus formulae
          [%new [<formula>]                          // Precondition
          [{ <update>* }]                            // Report messages
          [%retain <formula>]}                       // New individual(s)
<message> ::= (<quoted_string>|<pred>)⋈+             // Update formulae
<set_expr> ::= <set_name> | { <id>⋈, }               // Retain formula
             | <set_expr> - <set_expr>               // Message for user
             | <set_expr> + <set_expr>
<update> ::= <pred> ( <var>⋈, ) = <formula>          // Set difference
<formula> ::= <formula> & <formula>                  // Set union
             | <formula> | <formula>                 // Update formula
             | <formula> -> <formula>                // logical ∧
             | <formula> <-> <formula>               // logical ∨
             | !<formula>                            // logical implication
             | (<formula> ? <formula> : <formula>)   // logical equivalence
             | <var> == <var>                        // logical ¬
             | <var> != <var>                        // if-then-else
             | A( <var>⋈, )<formula>                 // equality
             | E( <var>⋈, )<formula>                 // inequality
             | <pred>( <var>⋈, )                     // ∀v1, v2, ..., vn
                                                     // ∃v1, v2, ..., vn
             | <pred>+( <var> , <var> )              // Predicate (of arbitrary
                                                     // arity)
             | <pred>*( <var> , <var> )              // Transitive closure on
                                                     // binary predicate
             | TC( <var> , <var> )                   // Reflexive and transitive
                  ( <var> , <var> ) <formula>        // closure on binary predicate
             | <kleene>                              // Transitive closure on a
<cfg_edge> ::= <cfg_node>                            // general binary formula
               <id> ( <id>⋈, ) <cfg_node>            // Atomic values
                                                     // CFG edge
```



Figure 6: The syntax of a TVP file.

| Property | Arity | Meaning | Consistency Rule |
|---|---|---|---|
| **unique** | unary | true for at most one node | p(v1) & p(v2) ==> v1 == v2<br>E(v1) p(v1) & v1 != v ==> !p(v) |
| **function** | binary | partial function | E(v) p(v, v1) & p(v, v2) ==> v1 == v2<br>E(v) p(v1, v) & v2 != v ==> !p(v1, v2) |
| **invfunction** | binary | inverse of a partial function | E(v) p(v1, v) & p(v2, v) ==> v1 == v2<br>E(v) p(v, v2) & v1 != v ==> !p(v1, v2) |
| **symmetric** | binary | | p(v1, v2) ==> p(v2, v1) |
| **antisymmetric** | binary | | p(v1, v2) & p(v2, v1) ==> v1 == v2<br>p(v1, v2) & v1 != v2 ==> !p(v2, v1) |
| **reflexive** | binary | | v1 == v2 ==> p(v1, v2) |
| **antireflexive** | binary | | v1 == v2 ==> !p(v1, v2) |
| **transitive** | binary | | E(v2) p(v1, v2) & p(v2, v3) ==> p(v1, v3) |
| **abs** | unary | $p$ is an abstraction predicate | N/A |
| **nonabs** | unary | $p$ is not an abstraction predicate | N/A |

Table 1: Properties of predicate p, their meaning and the generated consistency rules.

| Head | Condition | Result |
|---|---|---|
| 0 | | The structure is invalid - discard. |
| 1 | Never breached. | |
| predicate | The value of the predicate for the assignment is false<br>The value of the predicate for the assignment is unknown | The structure is invalid - discard.<br><br>Coerce it to true. |
| negated predicate | The value of the predicate for the assignment is true<br>The value of the predicate for the assignment is unknown | The structure is invalid - discard.<br><br>Coerce it to false. |
| variable equality | The two variables are assigned to different nodes<br>The variables are assigned to the same node and it is a summary node | The structure is invalid - discard.<br><br>Coerce into a non summary node. |
| variable inequality | The two variables are assigned to the same node | The structure is invalid - discard |

Table 2: Result of a consistency rule breach according to its head.

### 3.3 Consistency rules

Most of the needed consistency rules for an analysis are automatically generated from the functional properties of predicates (see Table 1) and from the instrumentation predicates' defining formulae. Sometimes it is useful to write explicit consistency rules. The left hand side of a consistency rule (the body) is a general formula, the right hand side (the head) is either an atomic formula or the negation of an atomic formula, `==>` stands for $\triangleright$. Note that the free variables of the body must match the free variables of the head exactly. A consistency rule state that for each assignment to the free variables of the body that evaluate the body to 1, the head should also evaluate to 1. The action performed in case of a consistency rule breach (i.e., the body of the consistency rule is evaluated to 1 and the head to 0 or 1/2 for a certain assignment) depends on the head of the consistency rule as seen in Table 2.

### 3.4 Actions

The arguments of an action are predicate names[1] that can be used in the following formulae and will be replaced with the actual arguments when the

[1] the arguments can also include identifiers that are used to define predicates.

action is used (see Section 3.5). The actions section of the reverse program is given in Figure 4.

### 3.4.1 Specifying the title of an action with %t

The title of the action, used when printing the action's structures.

### 3.4.2 Updating predicates

Update formulae describe how predicates are updated as a result of an action. If a predicate does not have an update formula then its value before the action is retained. The formula is evaluated on the old structure with the exception that nodes and predicates added in the **%new** declaration are available. Note that update clauses are not comma separated.

### 3.4.3 Using preconditions for filtering with %p

The precondition formula is evaluated to check whether this action should be performed. If the formula is closed then a result of true or unknown triggers the application of this action. If the formula contains free variables then the action is performed for each assignment into these variables potentially satisfying the formula. The free variables can be used in the following formulae and have the expected assignment.

### 3.4.4 Focusing structures with %f

The focus formulae for this action. Applied before the precondition.

### 3.4.5 Reporting messages with %message

Messages that are reported to the user if the formula given is potentially satisfied.

### 3.4.6 Creating new nodes with %new

An optional unary formula can be supplied. If no formula is supplied then a single new node is created. If a formula is supplied then each node potentially satisfying the formula is duplicated, a new temporary binary predicate called *instance* is created matching the old node with the new node. In both cases an unary predicated called $isNew$ is created an set true only for the nodes created in this action. Both these predicates can be used in the following formulae. The default value of all the predicates when applied to the new nodes is false. If the unary formula supplied evaluates to unknown for a certain node, the matching new node becomes maybe active.

### 3.4.7 Retaining nodes with %retain

A mechanism for defining nodes that persist after an action. By default, all nodes persist. This mechanism can be used to model actions like free or even on-the fly garbage collection. An unary formula must be supplied. Only nodes that potentially satisfy the formula are retained. If the formulae supplied evaluates to unknown for a certain node, it becomes maybe active instead of being removed.

## 3.5 Specifying the control flow graph

The program to be analyzed is composed of CFG nodes with edges connecting between them and actions to be performed on these edges. A flow insensitive analysis can be done by using a single CFG node with actions on self loops. A CFG node is declared implicitly by the existence of incoming or outgoing CFG edges. The action used in the CFG edge must be predefined in the actions section. The actual arguments passed to the action substitute the formal arguments used in its definition.

If only a subset of the nodes should be printed the list of CFG nodes to print should be supplied as the last section. The default behavior is printing the structures available in each CFG node.

## 3.6 Usability features

TVP was designed to be written generically. Several constructs are used to support this notion.

### 3.6.1 Writing comments

TVP supports C++ style comments: everything between /* and */ or from // to the end of that line is ignored.

### 3.6.2 Preprocessing

The TVP file can be preprocessed using the standard C preprocessor before being parsed by the system. The preprocessor enables file inclusion (using the #include directive), macro expansion (using the #define directive), and conditional evaluation (using the #if, #endif, etc. directives).

### 3.6.3 Sets

Sets are a mechanism for grouping together several predicate names to be used later in a **foreach** clause or a composite formula. Set operation such as union (+), and subtraction (−) can be used to create set expressions.

### 3.6.4 Foreach

Sometimes a declaration, a focus formula or an update formula should be repeated several times for different predicates, to avoid code duplication TVP support the mechanism of **foreach**. The syntax is:
**foreach (**<id> **in** <set_expr> **){**code **}**

The code between the curly braces is duplicates once for each set member and each time the predicate is substituted with the appropriate set member. Foreach can be applied to any declaration (core predicate, instrumentation predicate, consistency rule), to focus formulae and to update formulae in actions.

The **foreach** mechanism can handle composite predicate names, in this case only the identifiers within the square braces are substituted.

### 3.6.5 Composite operations

Composite operations are a mechanism for applying a logical operation (only **&** and **|** are supported) on a set of formulae. This is similar to **foreach** but can be used inside a formula. The syntax is:
<op>/{<formula> : <pred> **in** <set_expr>}

If the set is empty the neutral member for the operation is used (0 for **|** and 1 for **&**). For example, the expression **|**/{$z$(v) : $z$ **in** {$x$, $y$, $t$} } is expanded to $x$(v) **|** $y$(v) **|** $t$(v).

# 4 TVS

The input structures for the analysis are described in a format called TVS (Three Valued Structure). For example, the TVS for input structure used in the analysis of the reverse function is given in Figure 2. A TVS file name should end with the extension '.tvs'. The syntax of a TVS file is given in Figure 7.

The value of a predicate defaults to false unless otherwise specified in the TVS structure. All the node names used in the predicates must be predefined. All the predicate names used must be declared in the TVP file. If a node (or a node pair) is specified without a value, the default of true (1) is taken. TVS supports the same commenting style as TVP.

<tvs> ::= <structure>*
<structure> ::= <universe> <predicates>
<universe> ::= **%n = {** <node>⋈**, }**
<predicates> ::= **%p = {** <predicate>* **}**
<predicate> ::= <pred> **=** <kleene> /* Nullary */
| <pred> **= {** (<node> [<value>])⋈**, }** /* Unary */
| <pred> **= {** (<leftnode>**->**<rightnode> [<value>])⋈**, }** /* Binary */
| <pred> **= {** (<id>⋈**,**) [<value>]⋈**, }** /* Arbitrary */
<node> ::= <id>
<value> ::= **:** <kleene>

Figure 7: The syntax of a TVS file.

# 5 Command line options

```
Usage: tvla <program name> [input file] [options] Options:
 -d                        Turns on debug mode.
 -action [f][c]pu[c]b      Determines the order of operations computed
                           by an action. The default is fpucb.
                           f - Focus,  c - Coerce, p - Precondition.
                           u - Update, b - Blur.
 -join [algorithm]         Determines the type of join method to apply.
                           rel  - Relational join.
                           part - Partial join.
                           ind  - Independent attributes (single structure).
                           no_abstraction - No abstraction is applied.
 -ms <number>              Limits the number of structures.
 -mm <number>              Limits the number of messages.
 -save back|ext|all    Determines which locations store structures.
                           back - at every back edge (the default).
                           ext  - at every beginning of an extended block.
                           all  - at every program location.
 -noautomatic              Supresses generation of automatic constraints.
 -props <file name>        Can be used to specify a properties file.
 -log <file name>          Creates a log file of the execution.
 -tvs <file name>          Creates a TVS formatted output.
 -dot <file name>          Creates a DOT formatted output.
 -D<macro name>[(value)] Defines a C preprocessor macro.
 -terse                    Turns off on-line information printouts.
 -nowarnings               Causes all warnings to be ignored.
 -path <directory path>  Can be used to specify a search path.
```

```
-post                    Post order evaluation of actions.
```

- **Analysis engine**
  Three different types of engines are available : **tvla** is the classic chaotic iteration algorithm and the default one, **tvmc** is a multithreading engine that performs a state-space exploration using a search stack, and **ddfs** is a Double-DFS multithreading engine that utilizes Buchi automata.

- **Backward Analysis**
  Some analyses need to propagate information in the opposite direction specified for the CFG edges. To reverse the direction of the CFG edges use the **-backward** flag. When this option is chosen, the input structures are stored in the last location computed by the topological sorting of the program locations.

- **Order of action evaluation**
  The default order of evaluation in the iterative algorithm is reverse post order. However, when the analysis is very time/space consuming and you want to see the structures that reach the end of the analyzed program as soon as possible, use the **-post** flag to use post order and get the desired effect.

- **Debugging**
  When debugging a new analysis it is useful to see the analysis as it progresses and not just its final result. Use the **-d** flag to see the structures in the different phases of execution. In debug mode all the consistency rules are printed together with their dependencies and each time a structure is discarded because of an irreparable consistency rule breach, the problematic consistency rule, assignment and structure are shown. Notice that this mode generates very large PostScript files so you would probably want to use the **-ms** flag.

- **Computing the effect of an action**
  Sometime you want to try and run the algorithms (Coerce, Focus, Precondition, Update, Blur) in a different order or quantity than the default one (Focus, Coerce, Precondition, Update, Coerce, Blur). Use the **-action** <**seq**> flag to control the computation of the action's effect. The argument is of the form [f][c]pu[c]b when: f - Focus, c - Coerce, p - Precondition, u - Update, b - Blur.

- **Join method**
  Three join methods are available. To use the relational analysis approach where (bounded) structures kept up to isomorphism choose the

**rel** option. To use the single-structure method use the **ind** option. In this approach all the structures in a CFG node that match with their nullary predicates' values are merged into a single structure. The option is very useful for analyses that would otherwise take a very long time and create many structures. It is worth considering the **-action fcpucb** specification when working in single structure mode. A compromise between the two approaches merges structures that identify on their nullary abstraction values and set of canonical node names, thus considering only a partial set of predicates. To use this option choose the **part**. TVLA can perform analysis without applying any abstraction via the **no_abstraction** option. This option can yield a non-terminating analysis and therefore the **-ms** is usually needed to force termination.

- **Maximum number of structures**
  A complex analysis may take a very long time and especially in debug time may run forever. To see a partial result, you can limit the number of structures generated using the **-ms** <**number**> flag.

- **Maximum number of messages**
  The number of messages reported by the system can be limited. This can be used to supply a condition that if holds the system stops (by limiting the number of messages to 1).

- **Saved locations**
  The default behavior of the system is to perform a join, and save all the structures that reached the program location, only in every back edge in the control graph (approximately once in every loop). Saving structures at every program location is the most efficient in terms of the number of structures generated (use **-save all**). However, it is very space consuming. For a compromise between the two extremes use **-save ext** which saves structures only at the beginning of each extended block (i.e., at every merge point in the control graph).

- **No automatic constraint generation**
  Sometimes it is useful to supply all the constraints by hand without the automatically generated constraints. To do this supply the **-noautomatic** flag.

- **Properties file**
  A properties file name can be supplied with the **-props** option. The file is loaded before the analysis starts and overrides all other property files. For more information see section 6.

- **Log file**
  A log file name can be supplied with **-log** option. In this case the majority of the information written to the console is redirected to the log file.

- **TVS output file**
  A TVS output file name can be supplied with the **-tvs** option. In this case the output of the analysis in TVS format is directed to the file.

- **DOT output file**
  A DOT output file name can be supplied with the **-dot** option. In this case the output of the analysis in DOT format is directed to the file. The activation scripts for TVLA use this option to create an output file for DOT by adding the '.dt' extension to the program name. This option can be used to specify a different file name for DOT, but the Postscript output should then be created manually (the dotps script, supplied with TVLA, can be used for this).

- **On-line information**
  TVLA reports information as the analysis progresses to the standard output. To avoid seeing this information use the **-terse** flag.

- **Preprocessor macros**
  Preprocessor macros can be passed to the parsers by using the **-D<macro>(value)** option. This has the same effect as `#define symbol value`. For example, specifying -DCONCRETE defines the CONCRETE symbol, and -DLEVEL(1) sets the value 1 to the symbol LEVEL.

- **Search path**
  The option **-path** can be used to add directories to the search path.

# 6 Property files

A mechanism alternative to command-line options is available by the means of property files. Property files consist of key-value pairs using the syntax key=value. (More information about the syntax of property files is available from the JDK documentation of the java.util.Properties class[2].) Each command-line option has a corresponding property.

There are several ways to pass properties to TVLA:

[2] http://java.sun.com/j2se/1.4.2/docs/api/java/util/Properties.html

1. When TVLA is activated a tvla.properties file, which is located at the installation directory is loaded. This file contains all available properties and also includes documentation for each.

2. Though it is possible to edit the tvla.properties file, it is not recommended, because later TVLA versions may update this file and also because this global change will affect every run of TVLA. If global changes are intended, add the properties that should be change to the user.properties file, which is also located at the installation directory. The properties in this file override the ones in the tvla.properties file.

3. By creating a file with the same name as the program file and the .properties extension (for example merge.properties for merge.tvp) the properties in this file will load when this program is used and override the ones passed by the previous methods.

4. By specifying a properties file name with the -props command-line option. The properties in this file will override all properties specified with the previous methods.

5. Finally, command-line options change their corresponding properties and override properties specified by all other methods.

Although all command-line options have corresponding properties, there are other properties which do not correspond to command-line options. These are used to control TVLA behaviors which are less common and also to test features as they are being developed.

# 7 Changes since version 0.91

The following changes are incorporated:

- Two multi-threading engines are now available for analyzing multi-threaded programs. Currently, documentation for using them is only available from http://www.cs.tau.ac.il/~yahave/3vmc.htm.

- A properties mechanism is added. This is described in 6.

- The command-line options -b2 and -rotate have been moved to property file.

- The -significant option is no longer supported. This was done in order to offer better performance by adding different implementations for three-valued structures.

- The -dump option is no longer supported.
- The command-line option -join has been renamed to -save.
- The command-line option -single has been replaced with the -join option, which also includes a partial join (see the command-line options section for more details).
- The command-line option -action includes blur as a mandatory operation, which is applied last.
- The following command-line options are new : -props, -tvs, -dot, -D, -terse, and -nowarnings. Consult the command-line options sections before using them.
- It is now possible to specify empty predicate update sections.
- Nullary predicates can be presented as either diamonds or listed in a box. Use the tvla.dot.nullaryStyle property to choose the desired presentation.

# 8 Additional references

The TVLA system (version 0.9) was originally described in [**?**].

The TVLA framework is described in [**?**] and is a good place to start understanding the theory behind the system.

## References


[LA00] T. Lev-Ami. TVLA: A framework for Kleene based static analysis. Master's thesis, Tel-Aviv University, Tel-Aviv, Israel, 2000. 1

[LAS00] T. Lev-Ami and M. Sagiv. TVLA: A framework for Kleene based static analysis. In Jens Palsberg, editor, *Proc. Static Analysis Symp.*, volume 1824 of *Lecture Notes in Computer Science*, pages 280–301. Springer-Verlag, 2000. 1, 8

[SRW02] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3):217–298, 2002. 1, 8