TVLA: A Framework for Kleene Logic Based Static Analyses

Tal Lev-Ami*
Department of Computer Science, Tel-Aviv University, Israel

May 28, 2000

Acknowledgments

First and foremost I would like to thank Dr. Mooly Sagiv for his guidance, support and drive. Without it, this thesis would never have been written.

I would like to thank Nurit Dor, Manuel Fahndrich, Noam Rinetskey, Tom Reps, and Reinhard Wilhelm for reading the drafts and for their helpful comments. I enjoyed having valuable discussions with Hanne Riis and Flemming Nielson. Thanks also to Guy Laden, Ran Shaham and Oded Shmueli.

I would like to thank my parents Liora and Uzi for listening and giving a kind word where is was needed.

Many thanks to the Acadamy of Science for their Financial support.

Abstract

We present TVLA (Three-Valued-Logic Analyzer). TVLA is a "YACC"-like framework for automatically constructing static-analysis algorithms from an operational semantics, where the operational semantics is specified using logical formulae. TVLA has been implemented in Java and was successfully used to perform shape analysis on programs manipulating linked data structures (singly and doubly linked lists), to prove safety properties of Mobile Ambients, and to verify the partial correctness of several sorting programs.

*tla@math.tau.ac.il

Contents

1 Introduction

The abstract-interpretation technique [CC79] for static analysis allows one to summarize the behavior of a statement on an infinite set of possible memory 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 memory 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 [CC79] is usually not computable). This is particularly true in problems like shape analysis and pointer analysis (e.g., see [Deu94, SRW00, SRW98]), where the operational semantics involves destructive memory updates.

In this paper, we present TVLA (**T**hree-**V**alued-**L**ogic **A**nalyzer), a system for automatically generating a static-analysis algorithm from the operational semantics of a given program. The operational semantics is written in a special form, based on first-order predicate logic with transitive closure. An additional input to TVLA is an abstract representation of all the possible memory states at the beginning of the analyzed program. TVLA automatically generates the abstract semantics, and, for each program point, produces a conservative abstract representation of the memory states at that point.

1.1 Main Results

TVLA is intended as a proof of concept for intra-procedural shape analysis, and other static-analysis algorithms. It is a test-bed in which it is

quite easy to try out new ideas. The theory behind TVLA is based on [SRW99, SRW00] (see Section 8.2). The system is publicly available from `http://www.math.tau.ac.il/~tla`.

TVLA was implemented in Java and has been successfully used to perform shape analysis on programs manipulating linked data structures (singly and doubly linked lists), to prove safety properties of Mobile Ambients, and to verify partial correctness of several programs. We also report on some programs that are too complex for the current system. The system was tested on a Pentium II 400 MHz running Linux with JDK 1.2. All the timing information about the system refers to this computer[1].

1.1.1 Applications

TVLA has been utilized to analyze a variety of small but intricate programs from the groups described below.

Singly Linked Lists: We performed shape analysis on the set of programs manipulating singly linked lists used in [DRS00], including ones for searching, element insertion, and element deletion. These programs perform destructive updating. Some of these programs are (deliberately) semantically incorrect, and we are able to locate the bugs in them. The analysis times are reported in AppendixA.

Doubly Linked Lists: Doubly linked lists are more challenging than singly linked lists because they create shared memory cells and cycles. We have analyzed a program that inserts a new element into an arbitrary place in a doubly linked list, and the analysis was able to conclude that the insertion results in a doubly linked list.

Sorting Programs: A different kind of application of TVLA is for program verification. We applied TVLA to several implementations of sorting algorithms, and proved that, given a possibly unsorted linked list as input, we always end up with a sorted list. This is proven without the need for programmer-specified loop invariants. Instead, the operational semantics also keeps track of inequalities between the list elements. We are encouraged by the fact that we have successfully verified both insert sort and bubble sort on singly linked lists.

Mobile Ambients: We implemented the analysis of [NNS00] and found out that it is imprecise and quite slow. This motivated us to generalize the techniques presented in [SRW99, SRW00] in order to guarantee that only a constant number of structures arise at each program point (see Section 3.4).

[1] Our experience indicates that using JVM on Windows, the system runs about 20% faster.

With this extension, TVLA was able to successfully analyze a slight variant of the original specification used in [NNS00]. This took 336 CPU seconds and the analysis proved the necessary properties (uniqueness of ambient instance and mutual exclusion) precisely.

1.2 Technical Contributions

The TVLA system introduces several new contributions, which are described in this thesis.

Focus: We present a nontrivial algorithm for focusing on a general formula. Thus, unlike [SRW00], our Focus is not limited to the formulae specific to shape analysis. This generalization is crucial in order to go beyond shape analysis. For example, to verify sorting programs we use more complex formulae than the ones needed in shape analysis. The Focus algorithm presented in this thesis is also more efficient than the algorithm from [SRW99] for the formulae that they both handle.

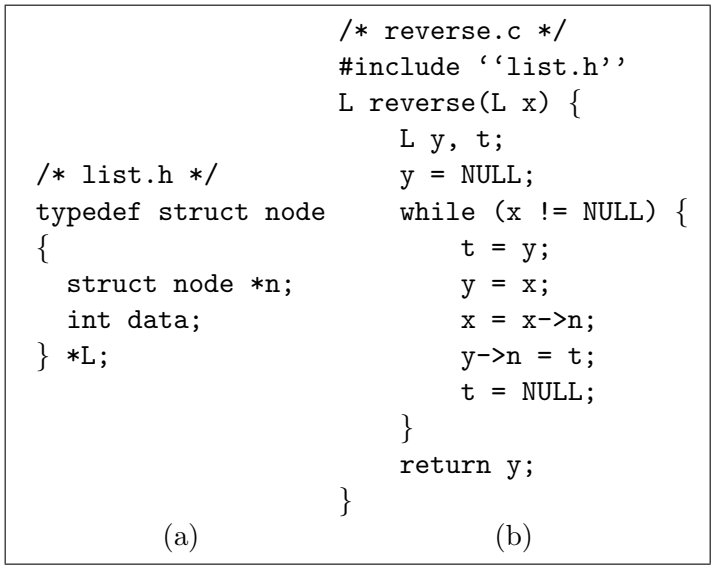
Coerce: The Coerce operation is very time consuming. TVLA introduces a new algorithm for Coerce (see Section 5) which is more efficient than the on given in [SRW00, SRW99], for empirical results, see Appendix A.

Automatic generation of consistency rules: One of the complicated aspects of using the three-valued logic approach is the design of consistency rules. This is particularly complicated because two logically equivalent sets of consistency rules may result in incomparable analyses (both of which are conservative). Furthermore, using consistency rules that are not global invariants may lead to an incorrect analysis. TVLA incorporates an algorithm that automatically generates consistency rules from the specification, and thus the user of the system does not usually need to add explicit consistency rules (see Section 5).

Constant number of structures: TVLA allows the use of an even more compact abstract representation in which only a constant number of abstract structures arise at each program point. In some cases (such as the analysis preformed on Mobile Ambients), this makes an otherwise infeasible analysis possible.

1.3 Outline of the Thesis

The rest of the thesis is organized as follows. In Section 2, we give a primer of the use of 3-valued logic in static analysis. Section 3 contains an overview of the TVLA system and its capabilities. Section 4 gives a description of the analyses done with the system. We then give a description of the



```
/* 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`.

main algorithms developed for the system: an efficient Coerce algorithm (Section 5) and a general Focus algorithm (Section 6). Section 7 explains the more advanced topic of active nodes. We conclude by summarizing related work and further research directions (Section 8). Appendix A presents the empirical results for test runs of the system. Appendix C is a user's manual for the TVLA system.

A program that destructively reverses a singly linked list is shown in Figure 1. The shape analysis of this program serves as a running example in this thesis.

2 A Primer on 3-Valued-Logic-Based Analysis

Kleene's 3-valued logic is an extension of ordinary 2-valued logic with the special value of 1/2 (unknown) for cases that can be either 1 or 0. Kleene's interpretation of the propositional operators is given in Table 1. We say that the values 0 and 1 are *definite values* and that 1/2 is an *indefinite value*. We say that the values 0 and 1 are *definite values* and that 1/2 is an *indefinite value*, and define a partial order $\sqsubseteq$ on truth values to reflect information

$\wedge$	0	1	1/2
0	0	0	0
1	0	1	1/2
1/2	0	1/2	1/2

$\vee$	0	1	1/2
0	0	1	1/2
1	1	1	1
1/2	1/2	1	1/2

$\neg$	
0	1
1	0
1/2	1/2

Table 1: Kleene's 3-valued interpretation of the propositional operators.

content: $l_1 \sqsubseteq l_2$ denotes that l_1 has more definite information than l_2:

Definition 2.1 [Information Order]. *For $l_1, l_2 \in \{0, 1/2, 1\}$, we define the* **information order** *on truth values as follows: $l_1 \sqsubseteq l_2$ if $l_1 = l_2$ or $l_2 = 1/2$. The symbol $\sqcup$ (join) denotes the least-upper-bound operation with respect to $\sqsubseteq$, i.e., $l_1 \sqcup l_2 = l_1$, if $l_1 = l_2$ and $1/2$ otherwise.*

Kleene's semantics of 3-valued logic is monotonic in the information order.

2.1 Representing Memory States via Logical Structures

Our vocabulary includes a set of predicate symbols partitioned into two disjoint sets: *core* and *instrumentation* predicates. Instrumentation predicates are used to observe derived properties based on core predicates.

A 2-*valued logical structure* S is comprised of a set of individuals (nodes) called a universe, denoted by U^S, and an interpretation over that universe for a set of predicate symbols. The interpretation of a predicate symbol p in S is denoted by p^S. For every (core and instrumentation) predicate p of arity k, p^S is a function $p^S \colon (U^S)^k \to \{0, 1\}$. 2-valued structures are used to represent memory states used in the operational semantics of the program.

TVLA makes an explicit assumption that the set of predicate symbols used throughout the analysis is fixed. (The number of individuals in structures can vary throughout the analysis.)

TVLA only supports predicates of arity ≤ 2; such logical structures can be thought of as directed graphs. A directed edge labeled by p from u_1 to u_2 denotes that $p^S(u_1, u_2) = 1$. Also, we draw p inside a node u when $p^S(u) = 1$.

Example 2.2 In the running example, a 2-valued structure represents a memory state (also called a *store*); an individual corresponds to a list element. The intended meaning of the core predicates is given in Table 2, and the intended meaning of the instrumentation predicates is given in Table 3

Predicate	Intended Meaning
$x(v)$	Is v pointed to by variable `x`?
$y(v)$	Is v pointed to by variable `y`?
$t(v)$	Is v pointed to by variable `t`?
$n(v_1, v_2)$	Does the `n`-field of v_1 point to v_2?

Table 2: The core predicates used in the analysis of the running example.

Predicate	Intended Meaning	Defining Formula
$r[n, x](v)$	Is v reachable from program variable `x` using field `n`?	$\exists v_1 : (x(v_1) \wedge n^*(v_1, v))$
$r[n, y](v)$	Is v reachable from program variable `y` using field `n`?	$\exists v_1 : (y(v_1) \wedge n^*(v_1, v))$
$r[n, t](v)$	Is v reachable from program variable `t` using field `n`?	$\exists v_1 : (t(v_1) \wedge n^*(v_1, v))$
$c[n](v)$	Does v reside on a directed cycle via dereferences along `n`-fields?	$n^+(v, v)$
$is[n](v)$	Is v pointed to by more than one `n`-field	$\exists v_1, v_2 : n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2$

Table 3: The instrumentation predicates used in the analysis of the running example and their meaning. Similar instrumentation predicates are used in all of our shape analyses for singly linked lists. The defining formulae are explained in Section 2.3.

(for the moment ignore the third column). The store in Figure 2 is represented by the 2-valued structure S_3 shown in Figure 3. The structure S_3 has four nodes, u_0, u_1, u_2, and u_3 representing the four list elements. This representation intentionally ignores the values of the data field, which are usually immaterial for the analysis.

Pointer variables are represented by unary predicates (i.e., $x^S(u) = 1$ if the variable `x` points to the list element represented by u). In Figure 3, the variable `x` is represented by the unary predicate x, which is 1 only for u_0. Notice that TVLA allows the user to specify that a unary predicate is drawn as a box with an arrow into each node for which it holds. In Figure 3, x is drawn as a box and has an arrow to u_0. Pointer fields within the list elements are represented as binary predicates (i.e., $n^S(u_1,\ u_2) = 1$ if the

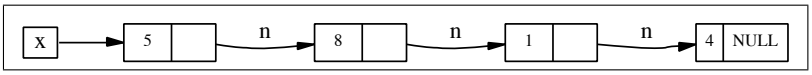
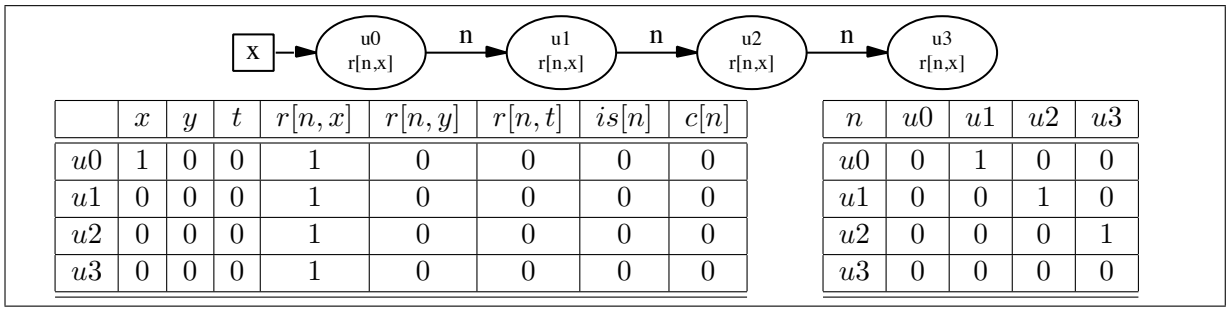


Figure 2: A possible store for the running example.



	x	y	t	$r[n,x]$	$r[n,y]$	$r[n,t]$	$is[n]$	$c[n]$
$u0$	1	0	0	1	0	0	0	0
$u1$	0	0	0	1	0	0	0	0
$u2$	0	0	0	1	0	0	0	0
$u3$	0	0	0	1	0	0	0	0

n	$u0$	$u1$	$u2$	$u3$
$u0$	0	1	0	0
$u1$	0	0	1	0
$u2$	0	0	0	1
$u3$	0	0	0	0

Figure 3: A logical structure S_3 representing the store shown in Figure 2 in a graphical and tabular representation.

`n`-field of u_1 points to u_2).

The instrumentation predicate $r[n,x]$ holds for list elements that are reachable from program variable `x`, possibly using a sequence of accesses through the `n`-field. The structure S_3 in Figure 3 has $r[n,x]^{S_3}$ set to 1 for all the nodes because they are all reachable from `x`. An important aspect of explicitly storing $r[n,x]$ is that we can incrementally compute the appropriate values for the predicates after execution of the program statement (see [SRW00, Section 6.1]). For example, for the statement `y = x`, the nodes reachable from `y` after the statement executes are the same as the nodes reachable from `x`.

The instrumentation predicate $is[n]$ holds for nodes shared by `n`-fields (a node is *shared* by `n`-fields, if it is pointed to by more than one list element using the field `n`). In Figure 3, all the elements of the list are unshared, and thus $is[n]^{S_3}$ is 0 for all of them.

The instrumentation predicate $c[n]$ holds for nodes on a cycle of accesses along `n`-fields. We use the cyclicity instrumentation to avoid performing a transitive-closure operation when updating the reachability information. In Figure 3, the list is acyclic, and thus $c[n]^{S_3}$ is 0 for all of the nodes.

In fact, throughout the analysis of the running example, $is[n]^S$ and $c[n]^S$ are 0 for all of the nodes.

2.2 Conservative Representation of Sets of Memory States via 3-valued Structures

Like 2-valued structures, a 3-*valued logical structure* S is also comprised of a universe U^S, and an interpretation p^S for every predicate symbol p. But, for every predicate p of arity k, p^S is a function $p^S\colon (U^S)^k \to \{0, 1, 1/2\}$, where $1/2$ explicitly captures unknown predicate values.

3-valued logical structures are also drawn as directed graphs. Definite values are drawn as in the 2-valued structures. Binary indefinite (1/2) predicate values are drawn as dotted directed edges. Unary indefinite predicate values are drawn inside the nodes and marked as indefinite (this does not occur in the running example).

2.2.1 Embedding

Although structures may have different individuals, we can define an order on structures, denoted by $\sqsubseteq$ based on the concept of *embedding*. The goal is to guarantee that if $S \sqsubseteq S'$ then the value of every formula in S is less or equal to its value in S'. In particular, whenever the formula evaluates to a definite value in S' then the formula has the same value in S. Formally,

Definition 2.3 *Let S and S' be two structures. Let $f\colon U^S \to U^{S'}$ be surjective. We say that f* **embeds** *S in S' (denoted by $S \sqsubseteq^f S'$) if (i) for every predicate p (including sm) of arity k and all $u_1, \ldots, u_k \in U^S$,*

$$p^S(u_1, \ldots, u_k) \sqsubseteq p^{S'}(f(u_1), \ldots, f(u_k)) \tag{1}$$

and (ii) for all $u' \in U^{S'}$

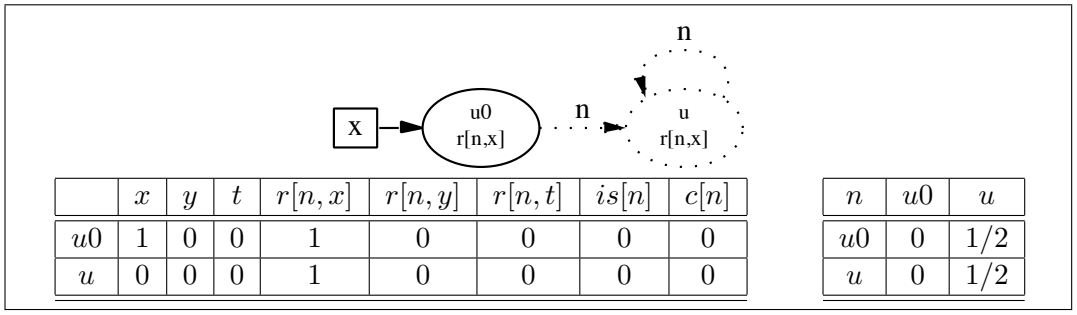
$$(|\{u \mid f(u) = u'\}| > 1) \sqsubseteq sm^{S'}(u') \tag{2}$$

We say that S **can be embedded in** *S' (denoted by $S \sqsubseteq S'$) if there exists a function f such that $S \sqsubseteq^f S'$.*

A special kind of embedding is a *tight embedding*, in which information loss is minimized when multiple individuals of S are mapped to the same individual in S':

Definition 2.4 *A structure S' is a* **tight embedding** *of S if there exists a surjective function $blur\colon U^S \to U^{S'}$ such that, for every $p \neq sm$ of arity k,*

$$p^{S'}(u'_1, \ldots, u'_k) = \bigsqcup_{blur(u_i)=u'_i, 1\leq i\leq k} p^S(u_1, \ldots, u_k) \tag{3}$$



	x	y	t	$r[n,x]$	$r[n,y]$	$r[n,t]$	$is[n]$	$c[n]$
$u0$	1	0	0	1	0	0	0	0
u	0	0	0	1	0	0	0	0

n	$u0$	u
$u0$	0	1/2
u	0	1/2

Figure 4: A 3-valued structure S_4 representing lists of length 2 or more that are pointed to by program variable `x` (e.g., S_3).

and for every $u' \in U^{S'}$,

$$sm^{S'}(u') = (|\{u | blur(u) = u'\}| > 1) \sqcup \bigsqcup_{blur(u)=u'} sm^{S}(u) \tag{4}$$

Because blur is surjective, equations (3) and (4) uniquely determine S' *(up to isomorphism); therefore, we say that* $S' = blur(S)$.

Example 2.5 In the running example, the 3-valued structure S_4 shown in Figure 4 represents the 2-valued structure S_3 for $f(u_0) = u_0$ and $f(u_1) = f(u_2) = f(u_3) = u$. In fact, the structure shown in Figure 4 represents all the lists with two or more elements.

The unary predicate symbol x has $x^{S_4}(u_0) = 1$, indicating that the program variable `x` is known to point to the list element represented by u_0, and $x^{S_4}(u) = 0$, indicating that `x` is known not to point to any of the list elements represented by u.

The binary predicate symbol n has $n^{S_4}(u_0, u) = 1/2$, indicating that the `n`-field of the list element represented by u_0 may point to a list element represented by u — namely the second list element (u_1 in Figure 3) — but does not point to all the list elements represented by u (e.g. u_2 in Figure 3). Also, $n^{S_4}(u, u) = 1/2$, indicating that the `n`-field of a list element represented by u may point to another list element represented by u or even to itself but does not point to all the list elements represented by u (e.g., in Figure 3 the `n`-field of u_2 points to u_3, but not to u_1).

2.2.2 Summary nodes

Nodes in a 3-valued structure that may represent more than one individual from a given 2-valued structure are called *summary nodes*. For example, in the structure shown in Figure 3, the nodes u_1, u_2, and u_3 are represented by the single node u in Figure 4.

TVLA uses a special designated unary predicate sm to maintain summary-node information. Such a summary node w has $sm^S(w) = 1/2$, indicating that it may represent more than one node in the embedded 2-valued structures. These nodes are graphically drawn as dotted ellipsis. In contrast, if $sm^S(w) = 0$ then w is known to represent a unique node. Only nodes with $sm^S(w) = 1/2$ can have more than one node mapped to them by the embedding function.

The exact choice of which nodes should be summarized is crucial for the precision of the analysis and is discussed in Section 2.2.1.

2.3 Formulae

Properties of structures can be extracted by evaluating formulae. We use first-order logic with transitive closure and equality, but without function symbols and constant symbols. For example, the formula

$$\exists v_1 : (x(v_1) \wedge n^*(v_1, v)) \tag{5}$$

extracts reachability information. Here, n^* denotes the reflexive transitive closure of the predicate n. Therefore, in every structure S, $x(v_1)$ evaluates to 1 if v_1 is the node pointed to by `x` and $n^*(v_1,\ v)$ evaluates to 1 in S if there exists a path of zero or more `n`-edges from v_1 to v. The third column of Table 3 displays the defining formula of all the instrumentation predicates used in the running example.

2.3.1 Subclasses of formulae

Atomic formulae are one of the following (i) $p(v_1, \ldots, v_k)$, (ii) $v_1 = v_2$, and (iii) 0 or 1. Without loss of generality only nullary, unary, and binary predicates are supported. A *literal* is an atomic formula or a negation of an atomic formula.

Definition 2.6 *A* **Horn clause** *is a formula of the form*

$$(\bigwedge_{i=1}^{m-1} \varphi_i) \rightarrow \varphi_m,$$

where $m > 1$, and φ_i is an atomic formula,

We now generalize the definition. Toward this end, for a formula φ, we define $\varphi^1 \equiv \varphi$ and $\varphi^0 \equiv \neg\varphi$. An **extended Horn clause** *is a formula φ of the form*

$$(\bigwedge_{i=1}^{m-1} (\varphi_i)^{B_i}) \rightarrow (\varphi_m)^{B_m},$$

where $m > 1$, φ_i is an atomic formula, $B_i \in \{0, 1\}$.

2.3.2 Semantics

Definition 2.7 *An* **assignment** *Z is a function that maps free variables to individuals (i.e., an assignment has the functionality $Z\colon \{v_1, v_2, \ldots\} \rightarrow U^S$). An assignment that is defined on all free variables of a formula φ is called* **complete** *for φ. In the sequel, we assume that every assignment Z that arises in connection with the discussion of some formula φ is complete for φ.*

The **meaning** *of a formula φ, denoted by $[\![\varphi]\!]^S(Z)$, yields a truth value in $\{0, 1, 1/2\}$. The meaning of φ is defined inductively as follows:*

Atomic *For a logical literal $\boldsymbol{l} \in \{\mathbf{0}, \mathbf{1}, \mathbf{1/2}\}$, $[\![\boldsymbol{l}]\!]^S(Z) = l$ (where $l \in \{0, 1, 1/2\}$).*

For an atomic formula $p(v_1, \ldots, v_k)$,

$$[\![p(v_1, \ldots, v_k)]\!]^S(Z) = p^S(Z(v_1), \ldots, Z(v_k))$$

For an atomic formula $(v_1 = v_2)$,

$$[\![v_1 = v_2]\!]^S(Z) = \begin{cases} 0 & Z(v_1) \neq Z(v_2) \\ 1 & Z(v_1) = Z(v_2) \text{ and } sm^S(Z(v_1)) = 0 \\ 1/2 & \text{otherwise} \end{cases}$$

Logical Connectives *For logical formulae φ_1 and φ_2*

$$\begin{aligned} [\![\varphi_1 \wedge \varphi_2]\!]^S(Z) &= \min([\![\varphi_1]\!]^S(Z), [\![\varphi_2]\!]^S(Z)) \\ [\![\varphi_1 \vee \varphi_2]\!]^S(Z) &= \max([\![\varphi_1]\!]^S(Z), [\![\varphi_2]\!]^S(Z)) \\ [\![\neg\varphi_1]\!]^S(Z) &= 1 - [\![\varphi_1]\!]^S(Z) \end{aligned}$$

Quantifiers *If φ is a logical formula,*

$$\begin{aligned} [\![\forall v_1 : \varphi]\!]^S(Z) &= \min_{u \in U^S} [\![\varphi_1]\!]^S(Z[v_1 \mapsto u]) \\ [\![\exists v_1 : \varphi]\!]^S(Z) &= \max_{u \in U^S} [\![\varphi_1]\!]^S(Z[v_1 \mapsto u]) \end{aligned}$$

Transitive Closure *For* $(TC\ v_1, v_2 : \varphi)(v_3, v_4)$,

$$[\![(TC\ v_1, v_2 : \varphi)(v_3, v_4)]\!]^S(Z) = \max_{\substack{n \geq 1, u_1, \ldots, u_{n+1} \in U, \\ Z(v_3) = u_1, Z(v_4) = u_{n+1}}} \min_{i=1}^{n} [\![\varphi]\!]^S(Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}])$$

We say that S *and* Z **potentially satisfy** φ *(denoted by* $S, Z \models \varphi$*) if* $[\![\varphi]\!]^S(Z) = 1/2$ *or* $[\![\varphi]\!]^S(Z) = 1$*. Finally, we write* $S \models \varphi$ *if for every* Z*:* $S, Z \models \varphi$*.*

The Embedding Theorem: The Embedding Theorem (see [SRW99, Theorem 3.7]) states that any formula that evaluates to a definite value in a 3-valued structure evaluates to the same value in all the 2-valued structures embedded into that structure. The Embedding Theorem is the foundation for the use of 3-valued logic in static-analysis: it ensures that it is sensible to reinterpret on the 3-valued structures the formulae, that when interpreted in 2-valued logic, define the operational semantics.

TVLA requires each instrumentation predicate to be associated with a formula over the core predicates defining its meaning. For example, evaluating formula (5) on the 3-valued structure shown in Figure 4, yields 1 for $v \mapsto u_0$, which indicates that the list element represented by u_0 is reachable from variable `x`, and 1/2 for $v \mapsto u$, which indicates that the list elements represented by u may or may not be reachable from program variable `x`. Notice that $r[n, x]^{S_4}(u) = 1$, which is more precise. This is a general principle with instrumentation predicates (referred to as the *instrumentation principle* in [SRW99]). The stored information can be more precise than the result of evaluating the corresponding formula.

3 System Description

The input to TVLA consists of two files: (i) a TVS (Three Valued logical Structure) file containing a textual representation of the input structures (see Figure 5), and (ii) a TVP (Three Valued Program) file, which includes the operational semantics and the association of the operational semantics with the edges of the control flow graph (CFG) of the analyzed program (see Figs. 6 and 7). To simplify the specification, we allow the operational semantics to be specific to the analyzed data type (e.g., singly linked lists in the running example). In the conversion of a C program into a TVP file, some

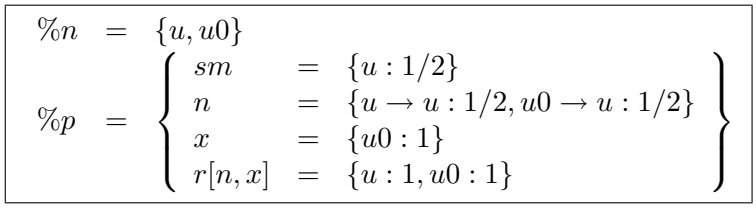


Figure 5: A TVS structure describing a singly linked list pointed to by `x`.

normalizing transformations are applied (see [CWZ90, SRW98]). For example, the assignment `y->n=t` is broken into two statements: (i) `y->n=NULL`, followed by (ii) `y->n=t` assuming that `y->n==NULL`. The full operational semantics for programs manipulating singly-linked-lists of type `L` is given in Section 4.1.

3.1 TVP

There are two challenging aspects to writing a good TVP specification: one is the design of the instrumentation predicates, which is important for the precision of the analysis; the other is writing the operational semantics manipulating these predicates.

An important observation is that the TVP specification should always be thought of in the terms of the concrete 2-valued world rather than the abstract 3-valued world: the Embedding Theorem guarantees the soundness of the reinterpretation of the formulae in the abstract world. This is an application of the well-known credo of Patrick and Radhia Cousot that the design of a static analysis always starts with a concrete operational semantics.

The TVP file is divided into sections separated by %%, given in the order described below.

3.1.1 Declarations

The first section of the TVP file contains all the declarations needed for the analysis.

Sets: The first declaration in the TVP file is the set *PVar*, which specifies the variables used in the program (here `x`, `y`, and `t`). In the remainder of the specification, set notation allows the user to define the operational semantics for all programs manipulating a certain data type, i.e., it is parametric in *PVar*.

```
/* Declarations */
%s PVar {x, y, t} // The set of program variables
#include "sll_pred.tvp" // Core and Instrumentation Predicates
%%
/* An Operational Semantics */
#include "ptr_cond.tvp" // Operational Semantics of Conditions
#include "sll_stat.tvp" // Operational Semantics of Statements
%%
/* The program's CFG and the effect of its edges */
n1 Set_Null_L(y) n2          // y = NULL;
n2 Is_Null_Var(x) exit       // x == NULL
n2 Is_Not_Null_Var(x) n3     // x != NULL
n3 Copy_Var_L(t, y) n4       // t = y;
n4 Copy_Var_L(y, x) n5       // y = x;
n5 Get_Next_L(x, x) n6       // x = x->n;
n6 Set_Next_Null_L(y) n7     // y->n = NULL;
n7 Set_Next_L(y, t) n8       // y->n = t;
n8 Set_Null_L(t) n2          // t = NULL;
```

Figure 6: The TVP file for the running example shown in Figure 1. Files sll_pred.tvp, sll_cond.tvp, and sll_stat.tvp are given in Figures 7, 10, and 11 respectively.

/* sll_pred.tvp */
foreach (z **in** PVar) {
  %p $z(v_1)$ **unique box** // Core predicates corresponding to program variables
}
%p $n(v_1, v_2)$ **function** // n-field core predicate
%i $is[n](v) = \exists v_1, v_2 : (n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2)$ // Is shared instrumentation
foreach (z **in** PVar) {
  %i $r[n, z](v) = \exists v_1 : (z(v_1) \wedge n^*(v_1, v))$ // Reachability instrumentation
}
%i $c[n](v) = \exists v_! : n(v, v_1) \wedge n^*(v_1, v)$ // Cyclicity instrumentation

Figure 7: The TVP predicate declarations for manipulating linked lists as declared in Figure 1 (a). The core predicates are taken from Table 2. Instrumentation predicates are taken from Table 3.

Predicates: The predicates for manipulating singly linked lists as declared in Figure 1(a) are given in Figure 7. The **foreach** clause iterates over all the program variables in the set *PVar* and for each of them defines the appropriate core predicate — the unary predicates x, y, and t (**box** tells TVLA to display the predicate as a box). The binary predicate n represents the pointer field `n`.

For readability, we use some mathematical symbols here that are written in C-like syntax in the actual TVP file (see [LA00, Appendix B]).

The second **foreach** clause (in Figure 7) uses *PVar* to define the reachability instrumentation predicates for each of the variables of the program (as opposed to Table 3, which is program specific). Thus, to analyze other programs that manipulate singly linked lists the only declaration that is changed is that of *PVar*.

The fact that the TVP file is specific for the data type L declared in Figure 1(a) allows us to explicitly refer to n.

Functional properties: TVLA also supports a concept of *functional properties* borrowed from the database community. Since program variables can point to at most one heap cell at a time, they are declared as **unique**. The binary predicate n represents the pointer field `n`; the `n`-field of each list element can only point to at most one target list element, and thus n is declared as a (partial) **function**.

3.1.2 Actions

In the second section of the TVP file, we define *actions* that specify the operational semantics of program statements and conditions. An action defines a 2-valued structure transformer. The actions are associated with CFG edges in the third section of the TVP file.

An action specification consists of several parts, each of which is optional (the meaning of these constructs is explained in Section 3.2). There are three major parts to the action: (i) Focus formulae (explained in Section 3.2.1), (ii) precondition formula specifying when the action is evaluated, and (iii) update formulae specifying the actual structure transformer. For example, the action `Is_Null_Var(x1)` (see Figure 10) specifies when the true branch of the condition `x1 == NULL`, is enabled by means of the formula $\neg\exists v : x1(v)$, which holds if `x1` does not point to any list element. Since this condition has no side effects there are no update formulae associated with this action and thus the structure remains unchanged. As another example, the action `Copy_Var_L(x1, x2)` (see Figure 11) specifies the semantics the statement `x1 = x2`. It has no precondition, and its side effect is to set the $x1$ predicate to $x2$ and the $r[n, x1]$ predicate to $r[n, x2]$.

3.1.3 CFG

The third section of the TVP specification is the CFG with actions associated with each of its edges. The edges are specified as **source action target**. The first CFG node that appears in the specification is the entry node of the CFG. The CFG specification for the running example, is given in Figure 6.

3.2 Process

This section presents a more detailed explanation, using the example shown in Figure 8, of how the effect of an action associated with a CFG edge is computed. To complete the picture, an iterative (fixed-point) algorithm to compute the result of static-analysis is presented in Section 3.3.

3.2.1 Focus

First, the Focus operation converts the input structure into a more refined set of structures that represents the same 2-valued structures as the input structure. Given a formula, Focus guarantees that the formula never evaluates to 1/2 in the focused structures. Focus (and Coerce) are semantic

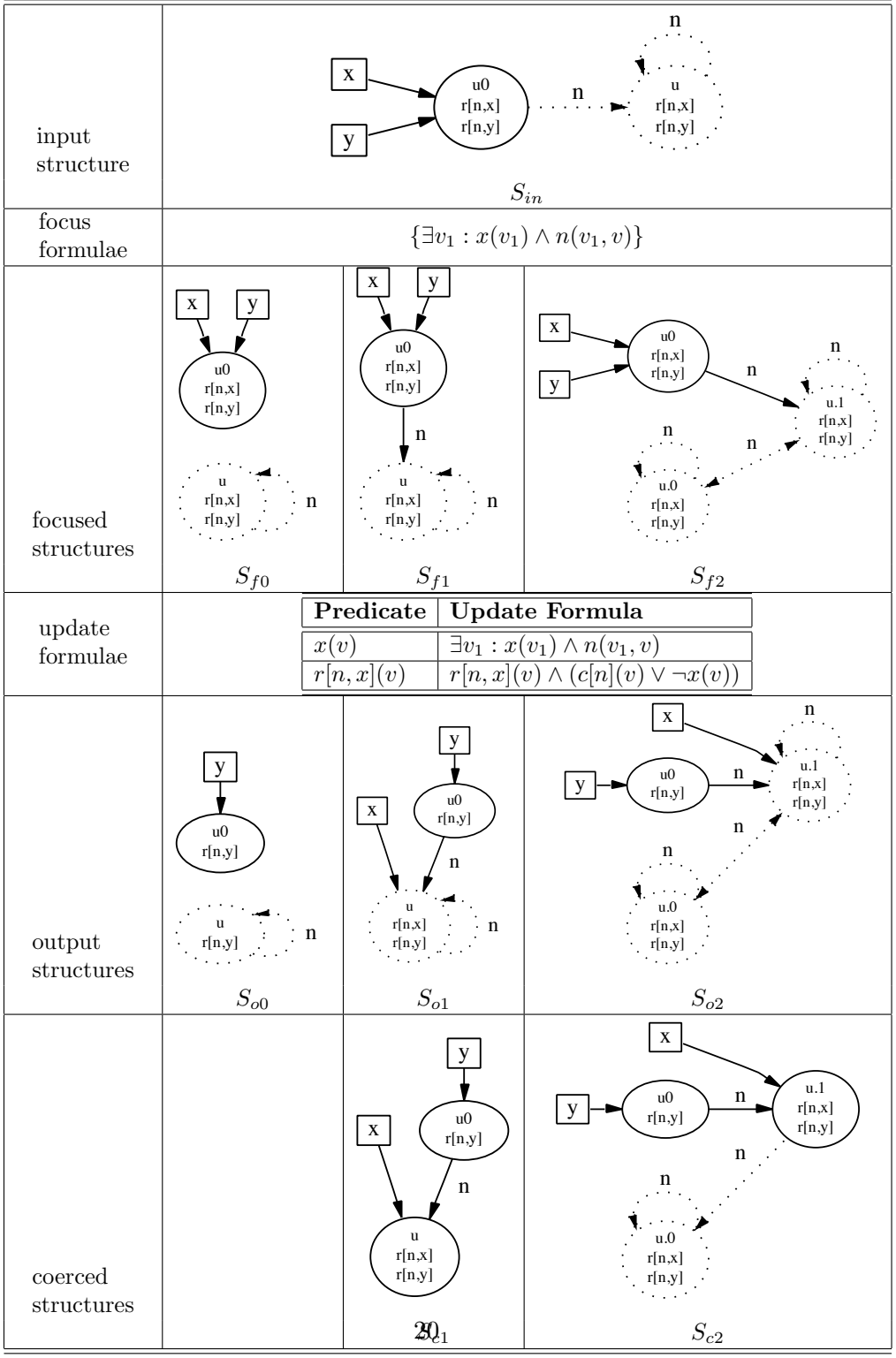


Figure 8: The first application of abstract interpretation for the statement `x = x->n` in the reverse function shown in Figure 1.

reductions (see [CC79]), i.e., they transfer a 3-valued structure into a set of 3-valued structures representing the same memory states. An algorithm for Focus of a general formula is given in [LA00]. In the running example, the most interesting focus formula is $\exists v_1 : x(v_1) \wedge n(v_1, v)$, which determines the value of the variable `x` after the `Get_Next_L(x, x)` action (which corresponds to the statement `x = x->n`). Focusing on this formula ensures that $x^S(u)$ is definite at every node u in every structure S after the action. Figure 8 shows how the structure S_{in} is focused for this action. Three cases are considered in refining S_{in}: (i) The `n`-field of u_0 does not point to any of the list elements represented by u (S_{f0}); (ii) The `n`-field of u_0 points to all of the list elements represented by u (S_{f1}); and (iii) The `n`-field of u_0 points to only some of the list elements represented by u (S_{f2}): u is bifurcated into two nodes — nodes pointed to by the `n`-field of u_0 are represented by $u.1$, and nodes not pointed to by the `n`-field of u_0 are represented by $u.0$.

As explained later, the result can be improved (e.g., S_{f0} can be discarded since u is not reachable from `x`, and yet $r[n, x]^{S_{f0}}(u) = 1$). This is solved by the Coerce operation, which is applied after the abstract interpretation of the statement (see Section 3.2.4).

3.2.2 Preconditions

After Focus, preconditions are evaluated. If the precondition formula is potentially satisfied, then the action is performed; otherwise, the action is ignored. This mechanism comes in handy for (partially) interpreting program conditions.

In the running example, the loop `while (x != NULL)` has two outgoing edges in the CFG: one with the precondition $\neg(\exists v : x(v))$, specifying that if `x` is NULL the statement following the loop is executed (the exit in our case). The other edge has the precondition $\exists v : x(v)$, specifying that if `x` is not NULL the loop body is executed.

3.2.3 Update Formulae

The effect of the operational semantics of a statement is described by a set of update formulae defining the value of each predicate after the statement's action. The Embedding Theorem enables us to reevaluate the formulae on the abstract structures and know that the result provides a conservative abstract semantics. If no update formula is specified for a predicate, it is left unchanged by the action.

In Figure 8, the effect of the `Get_Next_L` action (`x = x->n`) is computed using the following update formulae: (i) $x(v) = \exists v_1 : x(v_1) \wedge n(v_1, v)$, (ii) $r[n, x](v) = r[n, x](v) \wedge (c[n](v) \vee \neg x(v))$. The first formula updates the `x` variable to be the `n`-successor of the original `x`. The second formula updates the information about which nodes are reachable from `x` after the action: A node is reachable from `x` after the action if it is reachable from `x` before the action, except for the node directly pointed to by `x` (unless `x` appears on an `n`-cycle, in which case the node pointed to by `x` is still reachable even though we advanced to its `n`-successor). For S_{f2}, the update formula for x evaluates to 1 for $v \mapsto u.1$ and to 0 for all nodes other than $u.1$. Therefore, after the action, the resulting structure S_{o2} has $x^{S_{o2}}(u.1) = 1$ but $x^{S_{o2}}(u.0) = 0$ and $x^{S_{o2}}(u_0) = 0$.

3.2.4 Coerce

The last stage of the computation is the Coerce operation, which uses a set of consistency rules (defined in [SRW99, SRW00, LA00]) to make structures more precise by removing unnecessary indefinite values and discarding infeasible structures. The set of consistency rules used is independent of the current action being performed. See [LA00] for a detailed description of the Coerce algorithm used in TVLA and how TVLA automatically generated consistency rules from the instrumentation predicates and the functional properties of predicates.

For example, Figure 8 shows how the Coerce operation improves precision. The structure S_{o0} is infeasible because the node u must be reachable from `y` (since $r[n, y]^{S_{o0}}(u) = 1$) and this is not the case in S_{o0}. In the structure S_{o1}, u is no longer a summary node because `x` is **unique**; u's self-loop is removed because u already has an incoming `n`-field and it does not represent a shared list element ($is[n]^{S_{o1}}(u) = 0$). For the same reason, in S_{o2}, $u.1$ is no longer a summary node; Also, the list element represented by $u.1$ already has an incoming `n`-field and it is not shared ($is[n]^{S_{o2}}(u.1) = 0$), and thus $u.1$'s self-loop is removed. For a similar reason, the indefinite n-edge from $u.0$ to $u.1$ is removed.

3.2.5 Blur

To guarantee that the analysis terminates on programs containing loops, we require the number of potential structures for a given program to be finite.

Toward this end, we define the concept of a *bounded structure*. For each analysis, we choose a set of unary predicates called the *abstraction*

predicates.[2] In the bounded structure, two nodes u_1, u_2 are merged if $p^S(u_1) = p^S(u_2)$ for each abstraction predicate p. When nodes are merged, the predicate values for their non-abstraction predicates are joined (i.e., the result is 1/2 if their values are different). This is a form of widening (see [CC79]). The operation of computing this kind of bounded structure is called *Blur*. The choice of abstraction predicates is very important for the balance between space and precision. TVLA allows the user to select the abstraction predicates. By default, all the unary predicates are abstraction predicates, as in the running example.

Example 3.1 In Figure 4, the nodes u_0 and u are differentiated by the fact that $x^{S_4}(u_0) = 1$, whereas $x^{S_4}(u) = 0$. (All other predicates are 0.) If x was not an abstraction predicate, then the appropriate bounded structure S_4' would have had a single node, say u, with $x^{S_4'}(u) = 1/2$ and $n^{S_4'}(u, u) = 1/2$.

After the action is computed and Coerce applied, the Blur operation is used to transform the output structures into bounded structures, thereby generating more compact, but potentially less precise structures.

3.3 Output

Now that we have a method for computing the effect of a single action, what remains is to compute the effect of the whole program, i.e., to compute what structures can arise at each CFG node if the program was used on the given input structures. We use a standard iterative algorithm (e.g., see [Muc99]) with a set of bounded structures as the abstract elements. A new structure is added to the set if the set does not already contain a member that is isomorphic to the new structure. In the running example, the analysis terminates when the structures created in the fourth iteration are isomorphic to the ones created in the third iteration (see Figure 9). We can see that the analysis precisely captures the behavior of the reverse program.

3.4 Additional Features

The system allows several customizations on the standard iterative algorithm for optimizing the analysis (as command line options). The user can choose whether the actions are evaluated in depth first search post-order or reverse depth first search post-order. Even though reverse depth first search

[2] In [SRW99, SRW00] the abstraction predicates are all the unary predicates.

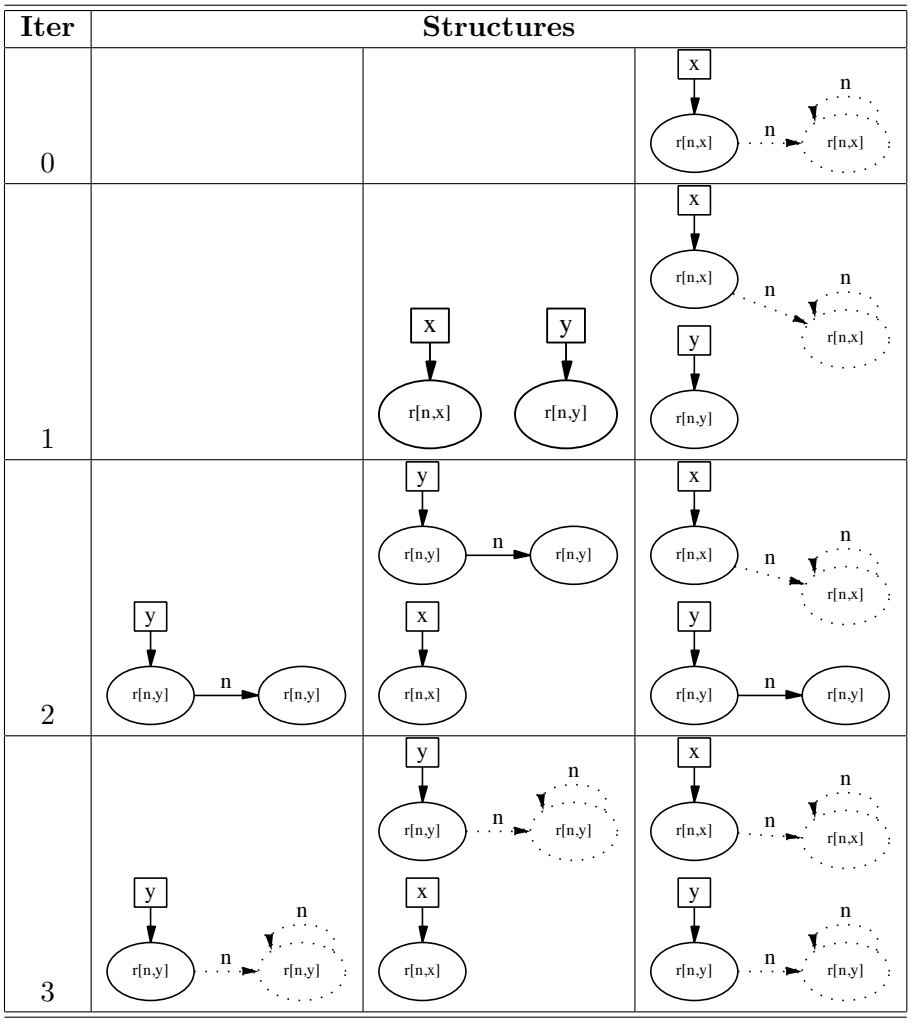


Figure 9: The structures arising in the reverse function shown in Figure 1 at CFG node n_2 for the input structure shown in Figure 4.

post order is usually more efficient (since an action is evaluated only after its predecessors are evaluated), using post order causes structures to reach the end of the program more quickly. This is very useful in case the analysis is not feasible and yet we want to see a glimpse of what is expected, an example for such an analysis is the merge function without reachability as defined in Section 4.1.

Another form of customization is the choice of CFG nodes in which the set of structures is saved. At the minimum at least one such node should reside on each loop in the CFG. Three forms are available: (i) at every CFG node, (ii) at every merge point (i.e., CFG node with two incoming edges), and (iii) at every back edge of the depth first search tree. The more CFG nodes in which the structures are saved the faster the analysis is (since structures need not be recreated). However, more space is needed (by factor of 10 even for simple programs).

One of the main features of TVLA is the support of single structure analysis. Sometimes when the number of structures that arise at each program point is too large, it is better to merge these structures into a single structure that represents at least the same set of 2-valued structures. TVLA enhances this feature even more by allowing the user to specify that some chosen constant number of structures will be associated with each program point.

More specifically, nullary predicates (i.e., predicates of 0-arity) are used to discriminate between different structures. For example, for linked lists we use the predicate $nn[x]() = \exists v : x(v)$ which discriminates between structures in which `x` actually points to a list element from structures in which it does not. For example, consider a structure S_1 in which both `x` and `y` point to list elements, and another structure S_2 in which both x and y are NULL. Merging S_1 and S_2 will loose the information that x and y are simultaneously allocated or not allocated. Notice that S_1 has $nn[x] = nn[y] = 1$ and S_2 has $nn[x] = nn[y] = 0$ therefore S_1 and S_2 will not be merged together.

In some cases (such as safety analysis of Mobile Ambients, see [NNS00]) this option makes an otherwise infeasible analysis run in a reasonable time. However, there are other cases in which the single-structure method is less precise or even more time consuming than the usual method, which uses sets of structures.

TVLA also supports modeling statements that handle dynamically allocated and freed memory.

```
/* ptr_cond.tvp */
%action Is_Not_Null_Var(x1) { %t x1 + " != NULL"
    %f { x1(v) } %p ∃v : x1(v)
}
%action Is_Null_Var(x1) { %t x1 + " == NULL"
    %f { x1(v) } %p ¬(∃v : x1(v))
}
%action Is_Eq_Var(x1, x2) { %t x1 + " == " + x2
    %f { x1(v), x2(v) }
    %p ∀v : x1(v) ⇔ x2(v)
}
%action Is_Not_Eq_Var(x1, x2) { %t x1 + " != " + x2
    %f { x1(v), x2(v) }
    %p ¬∀v : x1(v) ⇔ x2(v)
}
```

Figure 10: An operational semantics in TVP for handling pointer conditions.

4 Applications

4.1 Singly Linked Lists

We used the functions analyzed in [DRS00] with sharing and reachability instrumentations (see Table 4). The specification for all the functions was written once and used with each of the CFGs.

The actions for handling program conditions that consists of pointer equalities and inequalities are given in Figure 10.

The actions for manipulating the `struct node` declaration from Figure 1(a) are given in Figure 11. The actions `Set_Next_Null_L` and `Set_Next_L` model destructive updating (i.e., assignment to `x1->n`), and therefore have a nontrivial specification.

We use the notation $\varphi_1?\varphi_2 : \varphi_3$ for an if-then-else clause. If φ_1 is 1 then the result is φ_2, if φ_2 is 0 then the result is φ_3. If φ_1 is 1/2 then the result is $\varphi_2 \sqcup \varphi_3$. We use the notation $TC(v_1, v_2)(v_3, v_4)$ for the transitive-closure operator. The variables v_3 and v_4 are the free variables of the sub-formula over which the transitive closure is performed, and v_1 and v_2 are the variables used on the resulting binary relation.

Most of the analyses were very precise with running times of up to 8 seconds for the most complex function (merge).

```
/* sll_stat.tvp */
%action Set_Null_L(x1) { %t x1 + " = NULL"
    { x1(v) = 0    r[n, x1](v) = 0}
}
%action Copy_Var_L(x1, x2) { %t x1 + " = " + x2
    %f { x2(v) }
    { x1(v) = x2(v)    r[n, x1](v) = r[n, x2](v)}
}
%action Malloc_L(x1) { %t x1 + " = (L) malloc(sizeof(struct node)) "
    %new
    { x1(v) = isNew(v)    r[n, x1](v) = isNew(v) }
}
%action Free_L(x1) { %t "free(x1)"
    %f {x1(v)}
    %message ∃v1, v2 : x1(v1) ∧ n(v1, v2) ->
                "Internal error! assume that " + x1 + "->" + n + "==NULL"
    %retain ¬x1(v)
}
%action Get_Next_L(x1, x2) { %t x1 + " = " + x2 + "->" + n
    %f { ∃v1 : x2(v1) ∧ n(v1, v)}
    { x1(v) = ∃v1 : x2(v1) ∧ n(v1, v)
      r[n, x1](v) = r[n, x2](v) ∧ (c[n](v) ∨ ¬x2(v))}
}
%action Set_Next_Null_L(x1) { %t x1 + "->" + n + " = NULL"
    %f { x1(v) }
    { n(v1, v2) = n(v1, v2) ∧ ¬x1(v1)
      is[n](v) = is[n](v) ∧ (¬(∃v1 : x1(v1) ∧ n(v1, v))∨
                    ∃v1, v2 : (n(v1, v) ∧ ¬x1(v1)) ∧ (n(v2, v) ∧ ¬x1(v2)) ∧ v1 ≠ v2)
      r[n, x1](v) = x1(v)
      foreach(z in PVar-{x1}) {
          r[n, z](v) =(c[n](v) ∧ r[n, x1](v)?
                    z(v) ∨ ∃v1 : z(v1) ∧ TC(v1, v)(v3, v4)(n(v3, v4) ∧ ¬x1(v3)) :
                    r[n, z](v) ∧ ¬(r[n, x1](v) ∧ ¬x1(v) ∧ ∃v1 : r[n, z](v1) ∧ x1(v1)))
      }
      c[n](v) = c[n](v) ∧ ¬(∃v1 : x1(v1) ∧ c[n](v1) ∧ r[n, x1](v))}
}
%action Set_Next_L(x1, x2) { %t x1 + "->" + n + " = " + x2
    %f { x1(v), x2(v) }
    %message ∃v1, v2 : x1(v1) ∧ n(v1, v2) ->
                "Internal error! assume that " + x1 + "->" + n + "==NULL"
    { n(v1, v2) = n(v1, v2) ∨ x1(v1) ∧ x2(v2)
      is[n](v) = is[n](v) ∨ ∃v1 : x2(v1) ∧ n(v1, v)
      foreach(z in PVar) {
          r[n, z](v) = r[n, z](v) ∨ r[n, x2](v) ∧ ∃v1 : r[n, z](v1) ∧ x1(v1)
      }
      c[n](v) = c[n](v) ∨ (r[n, x2](v) ∧ ∃v1 : x1(v1) ∧ r[n, x2](v1))}
}
```

Figure 11: An operational semantics in TVP for handling the pointer-manipulation statements of linked lists as declared in Figure 1(a).

program	description
search	searches for an element in a linked list
null_deref	searches a linked list but with a typical error of not checking for the end of the list
delete	deletes a given element from a linked list
del_all	deletes an entire linked list
insert	inserts an element into a sorted linked list
create	prepend a varying number of new elements to a linked list
merge	merges two sorted linked lists into one sorted list
reverse	reverses a linked list via destructive updates
fumble	an erroneous version of reverse which loses the list
rotate	performs a cyclic rotation when given pointers to the first and last elements
swap	swaps the first and second elements of a list, fails when the list is 1 element long
getlast	returns the last element of the list

Table 4: Description of the analyzed singly linked list programs. These programs are collections of interesting programs from LCLint [Eva96], [JJNS97], Thomas Ball and from first-year students. They are available at `http://www.math.tau.ac.il/~nurr`.

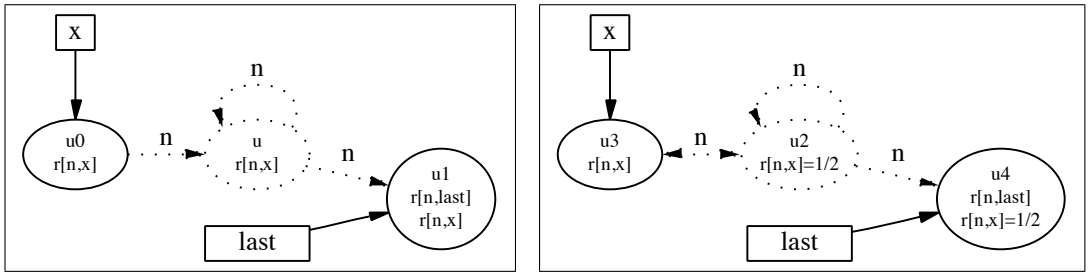


Figure 12: The structure before and after the rotate function.

The rotate function gives an example of an analysis which is not as precise as possible (see Figure 12). The indefinite edge $\langle [u.1, [u.0, u0].0],\ u1 \rangle$ is superfluous and all the list should be known to be reachable from x. The imprecision arises because the list becomes cyclic in the process and the reachability update formula in the action `Set_Next_Null_L` shown in Figure 11 is not very precise in case of cyclic lists.

The merge function is a good example of how precision problems in the analysis can create too many structures. Analyzing the merge function without the reachability instrumentation creates tens of thousands of graphs and takes too much space for the machine we were using. Adding the reachability information reduces the number of graphs to 327 and the time to about 8 seconds.

4.2 Doubly Linked Lists

Doubly linked lists are an example of a more complex abstract datatype which can still be analyzed accurately in many cases. The analysis is also a good example for how nuances in the instrumentation predicates used can change the accuracy of the analysis. We use a stronger instrumentation predicate than the one described in [SRW99], i.e., we keep a stronger program invariant. We show how using the new instrumentation increases the accuracy of the analysis. It is hard to predict how these small differences are going to affect the analysis, this demonstrates the importance of the system as a platform for developing and testing new analysis algorithms.

The splice function analyzed and the appropriate data structure are given in Figure 13. The TVP for splice is specified in figures 14, 16, and 15.

The instrumentation predicate that enables us to analyze the DLL programs precisely is the "cancel f by b" ($c[f, b]$), an unary predicate stating that `v->f->b == v`. A similar instrumentation is maintained for "cancel b by f". The formula used in [SRW99] for the instrumentation is,

$$c[f, b](v) = \forall v_1, v_2 : f(v, v_1) \wedge b(v_1, v_2) \rightarrow v_1 = v_2 \tag{6}$$

when trying to run the analysis on the splice function utilizing this defining formula, we found out that the analysis was not as precise as we wanted. We came up with the following definition of the instrumentation predicate,

$$c[f, b](v) = \forall v_1 : f(v, v_1) \rightarrow b(v_1, v) \tag{7}$$

The difference may seem insignificant. However, (7) is stronger than (6), i.e., a node that satisfies (7) must satisfy (6) but not vice versa. The constraints

```
/* dlist.h */
typedef struct DNode {
      struct DNode *f, *b;
      int data;
} DNode, *DL;
```

(a)

```
/* splice.c */
#include ‘‘dlist.h’’
void splice(int v, DL p) {
  DL e, t;
  e = (DL)malloc(sizeof(DNode));
  e->data = v;
  t = p->f;
  e->f = t;
  if  (t != NULL)
     t->b = e;
  p->f = e;
  e->b = p;
}
```

(b)

Figure 13: (a) Declaration of a doubly linked-list data type in C. (b) A program that splices an element with a data value `v` into a doubly linked list with a head pointed by `l`, after an element pointed to by `p`.

generated by the system from this instrumentation,

$$c[f, b](v) \wedge f(v, v_1) \triangleright b(v_1, v)$$

and,

$$c[f, b](v) \wedge \neg b(v_1, v) \triangleright \neg f(v, v_1)$$

are very important in keeping the analysis of the splice program precise. For example, Lets look at the simple $\mathbf{t = p\text{->}f}$ instruction with both versions of the instrumentation on the structure in Figure 17.

Figure 17 depicts a doubly linked list pointed to by `p`. Figure 18 illustrates a

4.3 Sorting Algorithms

In this section, we describe how the 3-valued-logic analysis framework can be used to prove that an implementation of an abstract datatype (ADT) is partially correct. Here we will be concerned with an ADT of sorted linked lists—i.e., a *subset* of the full set of data structures allowed according to the C typedef shown in Figure 1(a), consisting of those structures that

```
/* dll_pred.tvp */ %s PVar {l, p, t, e}
%s DSel {f, b}
/* Variables definition */
foreach (z in PVar) {
    %p z(v_1) unique box
}
foreach (sel in DSel) {
    /* Selector definition */
    %p sel(v_1, v_2) function
    /* Reachability instrumentation */
    foreach (z in PVar) {
        %i r[sel, z](v) = ∃v_1 : (z(v_1) ∧ sel*(v_1, v))
    }
    /* Cyclicity instrumentation */
    %i c[sel](v) = sel+(v, v)
    /* Cancel instrumentation */
    foreach (other in DSel-{sel}) {
        %i c[sel, other](v) = ∀v_1 : (sel(v, v_1) → other(v_1, v))
    }
}
```

Figure 14: The predicates used in analyzing doubly-linked lists declared as in Figure 13(a).

#include “dll_pred.tvp” // see Figure 14
%%
#include “ptr_cond.tvp” // see Figure 10
#include “dll_stat.tvp” // see Figure 16
%%
/* The program’s CFG
and the effect of its edges */
n_1 Malloc_DL(e) n_2
n_2 Get_Sel_DL(t, p, f) n_3
n_3 Set_Sel_Null_DL(e, f) n_4
n_4 Set_Sel_DL(e, f, t) n_5
n_5 Is_Not_Null_Var(t) n_6
n_5 Is_Null_Var(t) n_8
n_6 Set_Sel_Null_DL(t, b) n_7
n_7 Set_Sel_DL(t, b, e) n_8
n_8 Set_Sel_Null_DL(p, f) n_9
n_9 Set_Sel_DL(p, f, e) n_{10}
n_{10} Set_Sel_Null_DL(e, b) n_{11}
n_{11} Set_Sel_DL(e, b, p) *exit*

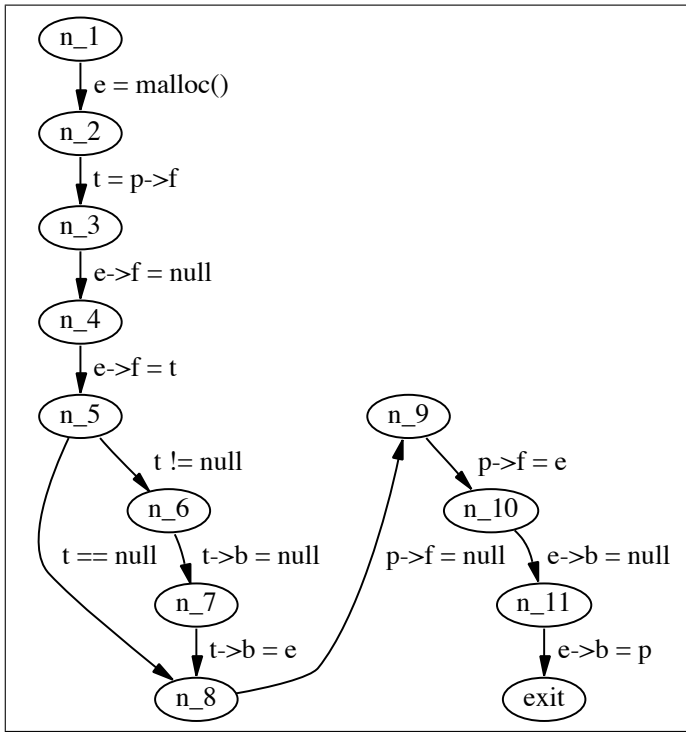


Figure 15: The CFG for the splice function shown in Figure 13.

```
/* dll_stat.tvp */
%action Get_Sel_DL(x1, x2, sel) { %t x1 + " = " + x2 + "->" + sel
    %f { ∃v1 : x2(v1) ∧ n(v1, v)}
    { x1(v) = ∃v1 : x2(v1) ∧ sel(v1, v)
      r[sel, x1](v) = r[sel, x2](v) ∧ (c[sel](v) ∨ ¬x2(v))
      foreach (other inPSel-{sel}) {
          r[other, x1](v) =(∃v1 : x2(v1) ∧ c[sel, other](v1)?
                            r[other, x2](v) ∨ ∃v1 : x2(v1) ∧ sel(v1, v) :
                            ∃v1, v2 : x2(v1) ∧ sel(v1, v2) ∧ other*(v2, v))
      } }
}
%action Malloc_DL(x1) { %t x1 + " = malloc() "
    %new
    { x1(v) = isNew(v)
      foreach (sel in DSel) {
          r[sel, x1](v) = isNew(v)
          foreach (other in DSel-{sel}) {
              c[sel, other](v) = c[sel, other](v) ∨ isNew(v)
          }
      } }
}
%action Set_Sel_Null_DL(x1, sel) { %t x1 + "->" + sel + " = null"
    %f { x1(v) }
    { sel(v1, v2) = sel(v1, v2) ∧ ¬x1(v1)
      r[sel, x1](v) = x1(v)
      foreach(z inPVar-{x1}) {
          r[sel, z](v) =(c[sel](v) ∧ r[sel, x1](v)?
                         z(v) ∨ ∃v1 : z(v1) ∧ TC(v1, v)(v3, v4)(sel(v3, v4) ∧ ¬x1(v3)) :
                         r[sel, z](v) ∧ ¬(r[sel, x1](v) ∧ ¬x1(v) ∧ ∃v1 : r[sel, z](v1) ∧ x1(v1)))
      }
      c[sel](v) = c[sel](v) ∧ ¬(∃v1 : x1(v1) ∧ c[sel](v1) ∧ r[sel, x1](v))
      foreach (other inDSel-{sel}) {
          c[sel, other](v) = x1(v) ∨ c[sel, other](v)
          c[other, sel](v) = c[other, sel](v) ∧ ¬∃v1 : x1(v1) ∧ other(v, v1)
      } }
}
%action Set_Sel_DL(x1, sel, x2) { %t x1 + "->" + sel + " = " + x2
    %f { x1(v), x2(v) }
    { sel(v1, v2) = sel(v1, v2) ∨ x1(v1) ∧ x2(v2)
      foreach(z in PVar) {
          r[sel, z](v) = r[sel, z](v) ∨ ∃v1 : r[sel, z](v1) ∧ x1(v1) ∧ r[sel, x2](v)
      }
      c[sel](v) = c[sel](v) ∨ (∃v1 : x1(v1) ∧ r[sel, x2](v1) ∧ r[sel, x2](v))
      foreach (other in DSel-{sel}) {
          c[sel, other](v) = (x1(v)?∃v1 : other(v1, v) ∧ x2(v1) : c[sel, other](v))
          c[other, sel](v) = c[other, sel](v) ∨ (x2(v) ∧ ∃v1 : other(v, v1) ∧ x1(v1))
      } }
}
```


Figure 16: The new actions defined for the splice function shown in Figure 13.

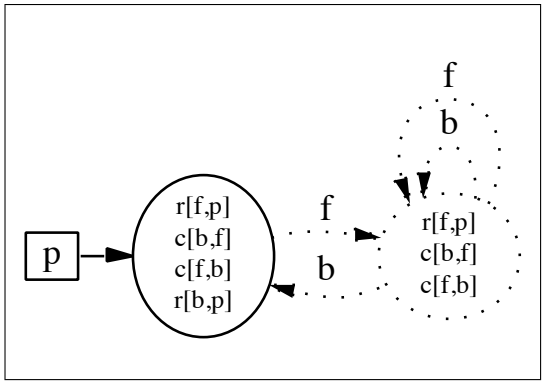


Figure 17: The structure before `t = p->f`.

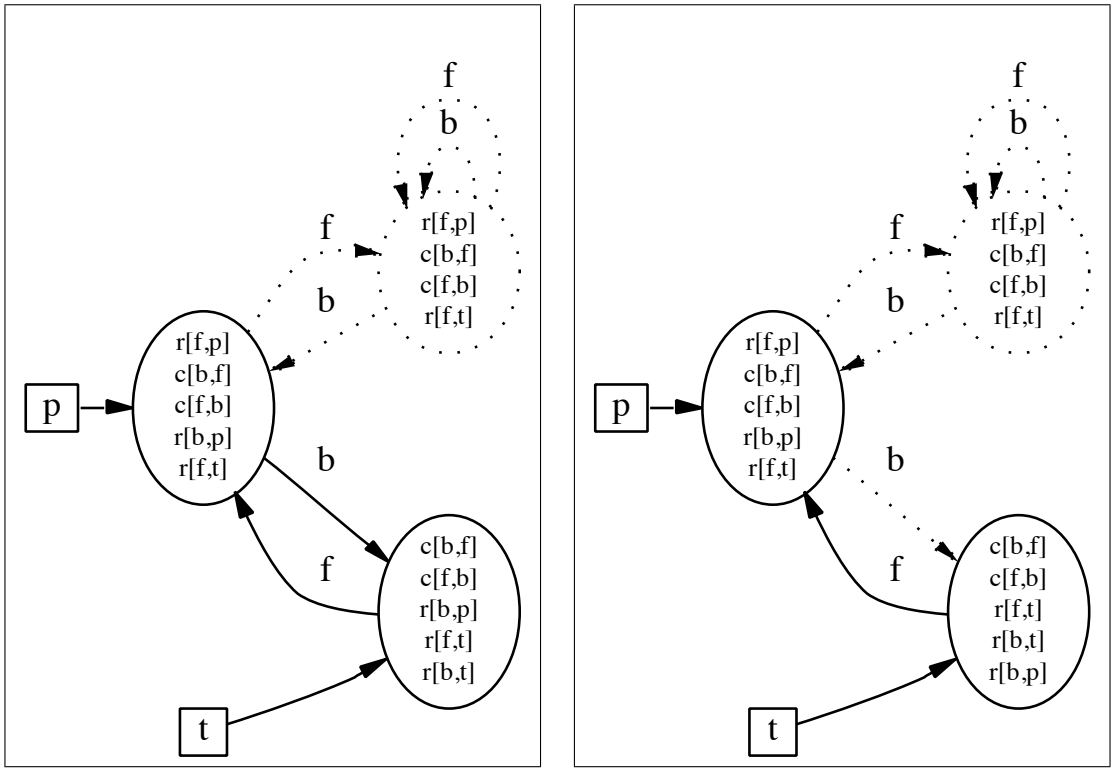


Figure 18: The structure after `t = p->f` with (a) the instrumentation presented here and (b) the instrumentation defined in [SRW99].

```
/* main.c */
#include "list.h"
int main() {
  L x, y, z, w;
  L create(), insert_sort(L);
  L merge(L,L), reverse(L);

  x = create(); l1:
  x = insert_sort(x); l2:

  y = create(); l3:
  y = insert_sort(y); l4:

  z = merge(x,y); l5:
  w = reverse(z); l6:
}
```

Figure 19: A main program that performs several operations on sorted lists.

meet the "is-sorted" datatype invariant. In the case of sorted linked lists, we are interested not just in the correctness of various sorting operations, which create sorted linked lists, but also in establishing that "is-sorted" is maintained by list-manipulation operations, such as element-insertion, element-deletion, destructive list reversal, and merging of two lists. In this section we refer to the code in Figure 19 to explain the analyses done.

A sorting procedure is *partially correct* if, whenever the procedure terminates, the output list it produces is sorted in non-decreasing order. Our approach to verification is capable of establishing this. For instance, the specific analysis that we discuss below establishes that at program point l_2 in Figure 19, program variable `x` always points to a list sorted in non-decreasing order (cf. Figure 27). It also establishes that at program point l_6, program variable `w`, which holds the reversal of the merge of two sorted lists, always points to a list sorted in non-increasing order (cf. Figure 30).

Verification will sometimes fail because the analysis is *conservative*, i.e., it may be that the analysis reports that, at a given program point, a variable might point to something other than a sorted list when in fact it always does

point to a sorted list. Our limited experience with several small but intricate programs indicates that this does not happen. The Embedding Theorem guarantees that the converse is impossible: The analysis can never say that at a given program point l, variable `x` always points to a sorted list, and yet there is an input that leads to a store at l in which the `x` list is not sorted. Thus, if the analysis says that at the exit vertex variable `x` always points to a sorted list, then `x` will always point to a sorted list when the procedure finishes execution.

An artifact of our approach is that some of the work involved in verification takes place at the level of the programming language, rather than at the level of individual statements of a program. Section 4.3.1 discusses what is required at the programming-language level to define a suitable analysis for observing ADT properties; Section 4.3.2 describes how such an analysis can then be used to check the partial correctness of ADT operations.

4.3.1 Specifying an Analysis for Observing ADT Properties

The static analysis algorithm used is the one depicted is Section 3. Predicate-update formulae for the core predicates x (for all $x \in PVar$) and n, along with definitions and predicate-update formulae for the instrumentation predicates is, c, and $r[n, x]$ can be found in Tables 3 and 2. To define an analysis suitable for verifying procedures that operate on sorted linked lists, we have to provide predicate-update formulae for the predicates dle, $inOrder[dle, n]$ and $inROrder[le, n]$ (for each of the statements that manipulate pointer variables), their definition is given in Figure 21. The actions specifying the operational semantics for the statements of the sorting programs is given in Figure 23, and the actions specifying the operational semantics for the conditions used in the sorting programs is given in Figure 22. For bubble_sort we use another ADT of a boolean variable. The actions needed to manipulate these kind of variables are given in Figure 25.

4.3.2 Specifying and Checking Partial Correctness of ADT Operations

Given the static-analysis algorithm defined in the preceding section, to demonstrate the partial correctness of ADT operations, the user must supply the following program-specific information:

- The procedure's control-flow graph.
- A set of 3-valued structures that characterize the acceptable inputs to the procedure.

```
/* insert_sort.c */
#include ‘‘list.h’’
void insert_sort(L x) {
  r = x;
  pr = NULL;
  while (r != NULL) {
    l = x;
    rn = r->n;
    pl = NULL;
    while (l != r) {
      if(l->data > r->data) {
        pr->n = rn;
        r->n = l;
        if(pl == NULL)
          x = r;
        else
          pl->n = r;
        r = pr;
        break;
      }
      pl = l;
      l = l->n;
    }
    pr = r;
    r = rn;
  }
}
```

(a)

%s PVar {x, r, pr, rn, pl, l}
#include “sorting_pred.tvp”
%%
#include “sorting_cond.tvp”
#include “sorting_stat.tvp”
%%

n_1 Copy_Var_L(r, x) n_2
n_2 Set_Null_L(pr) n_3
n_3 Is_Not_Null_Var(r) n_4
n_3 Is_Null_Var(r) *exit*
n_4 Copy_Var_L(l, x) n_5
n_5 Get_Next_L(rn, r) n_6
n_6 Set_Null_L(pl) n_7
n_7 Is_Not_Eq_Var(l, r) n_8
n_7 Is_Eq_Var(l, r) n_{20}
n_8 Greater_L(l, r) n_9
n_8 Less_Equal_L(l, r) n_{18}
n_9 Set_Next_Null_L(pr) n_{10}
n_{10} Set_Next_L(pr, rn) n_{11}
n_{11} Set_Next_Null_L(r) n_{12}
n_{12} Set_Next_L(r, l) n_{13}
n_{13} Is_Null_Var(pl) n_{14}
n_{13} Is_Not_Null_Var_L(pl) n_{15}
n_{14} Copy_Var_L(x, r) n_{17}
n_{15} Set_Next_Null_L(pl) n_{16}
n_{16} Set_Next_L(pl, r) n_{17}
n_{17} Copy_Var_L(r, pr) n_{20}
n_{18} Copy_Var_L(pl, l) n_{19}
n_{19} Get_Next_L(l, l) n_7
n_{20} Copy_Var_L(pr, r) n_{21}
n_{21} Copy_Var_L(r, rn) n_3

(b)

Figure 20: The C code for the `insert_sort` function (a) and its corresponding CFG.

```
/* sorting_pred.tvp */
#include "sll_pred.tvp"
/* Comparator definition */
%p dle(v1, v2) transitive reflexive
/* Ordering instrumentations */
%i inOrder[dle, n](v) = ∀v1 : n(v, v1) → dle(v, v1)
%i inROrder[dle, n](v) = ∀v1 : n(v, v1) → dle(v1, v)
```

Figure 21: The TVP declarations for the `insert_sort` function shown in Figure 20(a) (sll_pred.tvp is defined in Figure 7).

```
/* sorting_cond.tvp */
#include "sll_cond.tvp"

%action Greater_L(x1, x2) { %t x1 + "->d >" + x2 + "->d"
    %f { ∃v1, v2 : x1(v1) ∧ x2(v2) ∧ dle(v1, v2) }
    %p ¬∃v1, v2 : x1(v1) ∧ x2(v2) ∧ dle(v1, v2)
}
%action Less_Equal_L(x1, x2) { %t x1 + "->d <= " + x2 + "->d"
    %f { ∃v1, v2 : x1(v1) ∧ x2(v2) ∧ dle(v1, v2) }
    %p ∃v1, v2 : x1(v1) ∧ x2(v2) ∧ dle(v1, v2)
}
%action Equal_L(x1, x2) { %t x1 + "->d == " + x2 + "->d"
    %f { ∃v1, v2 : x1(v1) ∧ x2(v2) ∧ dle(v1, v2) ∧ dle(v2, v1) }
    %p ∃v1, v2 : x1(v1) ∧ x2(v2) ∧ dle(v1, v2) ∧ (dle(v2, v1)
}
```

Figure 22: The conditions needed for the `insert_sort` function shown in Figure 20. sll_cond.tvp is defined in Figure 10.

```
/* sorting_stat.tvp */
%action Set_Null_L(x1) { %t x1 + " = NULL"
    { x1(v) = 0
      r[n, x1](v) = 0}
}
%action Set_Next_Null_L(x1) { %t x1 + "->" + n + " = NULL"
    %f { x1(v) }
    { n(v1, v2) = n(v1, v2) ∧ ¬x1(v1)
      is[n](v) = is[n](v) ∧ (¬(∃v1 : x1(v1) ∧ n(v1, v))∨
                  ∃v1, v2 : (n(v1, v) ∧ ¬x1(v1)) ∧ (n(v2, v) ∧ ¬x1(v2)) ∧ v1 ≠ v2)
      r[n, x1](v) = x1(v)
      foreach(z in PVar-{x1}) {
          r[n, z](v) =(c[n](v) ∧ r[n, x1](v)?
                  z(v) ∨ ∃v1 : z(v1) ∧ TC(v1, v)(v3, v4)(n(v3, v4) ∧ ¬x1(v3)) :
                  r[n, z](v) ∧ ¬(r[n, x1](v) ∧ ¬x1(v) ∧ ∃v1 : r[n, z](v1) ∧ x1(v1)))
      }
      c[n](v) = c[n](v) ∧ ¬(∃v1 : x1(v1) ∧ c[n](v1) ∧ r[n, x1](v))
      inOrder[dle, n](v) = inOrder[dle, n](v) ∨ x1(v)
      inROrder[dle, n](v) = inROrder[dle, n](v) ∨ x1(v)}
}
%action Set_Next_L(x1, x2) { %t x1 + "->" + n + " = " + x2
    %f { x1(v), x2(v) }
    %message ∃v1, v2 : x1(v1) ∧ n(v1, v2) ->
              "Internal error! assume that " + x1 + "->" + n + "==NULL"
    { n(v1, v2) = n(v1, v2) ∨ x1(v1) ∧ x2(v2)
      is[n](v) = is[n](v) ∨ ∃v1 : x2(v) ∧ n(v1, v)
      foreach(z in PVar) {
          r[n, z](v) = r[n, z](v) ∨ r[n, x2](v) ∧ ∃v1 : r[n, z](v1) ∧ x1(v1)
      }
      c[n](v) = c[n](v) ∨ (r[n, x2](v) ∧ ∃v1 : x1(v1) ∧ r[n, x2](v1))
      inOrder[dle, n](v) = inOrder[dle, n](v) ∧ ¬(∃v1 : x1(v) ∧ x2(v1) ∧ ¬dle(v, v1))
      inROrder[dle, n](v) = inROrder[dle, n](v) ∧ ¬(∃v1 : x1(v) ∧ x2(v1) ∧ ¬dle(v1, v))}
}
%action Malloc_L(x1) { %tx1 + " = (L) malloc(sizeof(struct node)) "
    %new
    { x1(v) = isNew(v)
      r[n, x1](v) = isNew(v)
      dle(v1, v2) =
          (isNew(v1) ∧ isNew(v2))∨
          (v1 ≠ v2 ∧ (isNew(v1) ∨ isNew(v2))?1/2 : dle(v1, v2))
      inOrder[dle, n](v) = inOrder[dle, n](v) ∨ isNew(v)
      inROrder[dle, n](v) = inROrder[dle, n](v) ∨ isNew(v)
    }
}
%action Free_L(x1) { %t"free(" + x1 + ") "
    %f{ x1(v) }
    %retain¬x1(v)
}
```

Figure 23: The actions for the statements needed for the `insert_sort` function shown in Figure 20.

```
/* bubble_sort.c */
#include ‘‘list.h’’
void bubble_sort(L x) {
   change = TRUE;
   while (change) {
      p = NULL;
      change = FALSE;
      y = x;
      yn = y->n;
      while (yn != NULL) {
         if (y->data > yn->data) {
            t = yn->n;
            change = TRUE;
            y->n = t;
            yn->n = y;
            if (p == NULL)
               x = yn;
            else
               p->n = yn;
            p = yn;
            yn = t;
         } else {
            p = y;
            y = yn;
            yn = y->n;
         }
      }
   }
}
```

(a)

%s *PVar* {x, y, p, yn, t}
#include “sorting_pred.tvp”
%p change()
%%
#include “sorting_cond.tvp”
#include “sorting_stat.tvp”
#include “bool_cond.tvp”
#include “bool_stat.tvp”
%%
n_1 Set_True(change) n_2
n_2 Is_True(change) n_3
n_2 Is_False(change) n_{24}
n_3 Set_Null_L(p) n_4
n_4 Set_False(change) n_5
n_5 Copy_Var_L(y, x) n_6
n_6 Get_Next_L(yn, y) n_7
n_7 Is_Not_Null_Var_L(yn) n_8
n_7 Is_Null_Var_L(yn) n_2
n_8 Greater_L(y, yn) n_9
n_8 Less_Equal_L(y, yn) n_{21}
n_9 Get_Next_L(t, yn) n_{10}
n_{10} Set_True(change) n_{11}
n_{11} Set_Next_Null_L(y) n_{12}
n_{12} Set_Next_L(y, t) n_{13}
n_{13} Set_Next_Null_L(yn) n_{14}
n_{14} Set_Next_L(yn, y) n_{15}
n_{15} Is_Null_Var_L(p) n_{16}
n_{15} Is_Not_Null_Var_L(p) n_{17}
n_{16} Copy_Var_L(x, yn) n_{19}
n_{17} Set_Next_Null_L(p) n_{18}
n_{18} Set_Next_L(p, yn) n_{19}
n_{19} Copy_Var_L(p, yn) n_{20}
n_{20} Copy_Var_L(yn, t) n_7
n_{21} Copy_Var_L(p, y) n_{22}
n_{22} Copy_Var_L(y, yn) n_{23}
n_{23} Get_Next_L(yn, y) n_7
n_{24} Set_Null_L(p) n_{25}
n_{25} Set_Null_L(y) n_{26}
n_{26} Set_Null_L(yn) n_{27}
n_{27} Set_Null_L(t) *exit*

(b)

Figure 24: The C code for the `bubble_sort` function with element swap (a) and its corresponding TVP.

```
/* bool_stat.tvp */
%action Set_True(x1) { %t x1 + "=TRUE"
    { x1() = 1 }
}
%action Set_False(x1) { %t x1 + "=FALSE"
    { x1() = 0 }
}
```

```
/* bool_cond.tvp */
%action Is_True(x1) { %t x1
    %p x1()
}
%action Is_False(x1) { %t "!" + x1
    %p ¬x1()
}
```

Figure 25: Statements and conditions manipulating boolean variables.

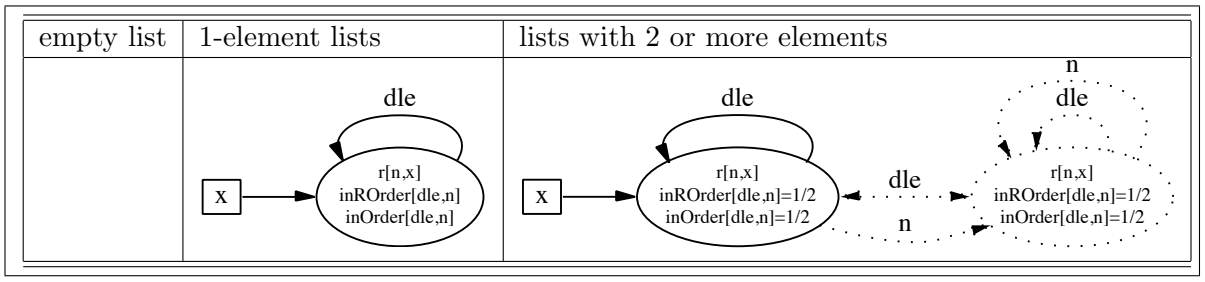


Figure 26: The structures that arise at l_1.

- Formulae that characterize the acceptable outputs of a correctly working procedure.

The initial 3-valued structures are supplied to the analysis algorithm as the abstract value for the procedure's entry point; the analysis algorithm is then run; finally, the formulae that characterize the acceptable outputs are applied to the structures that are generated by the analysis at the procedure's exit point.

Example 4.1 Consider the problem of establishing that the version of `insert_sort` shown in Figure 20 is partially correct. Figure 26 shows the three structures that characterize the set of stores in which program variable `x` points to an acyclic, unshared linked list.[3] After running the analysis of `insert_sort`, we would check to see whether, for all of the structures that arise at the procedure's exit node, the following formula evaluates to 1:

$$\forall v : r[n, x](v) \Rightarrow inOrder[dle, n](v). \tag{8}$$

[3]These are exactly the 3-valued structures that the analysis discovers as the possible outputs of `create`.

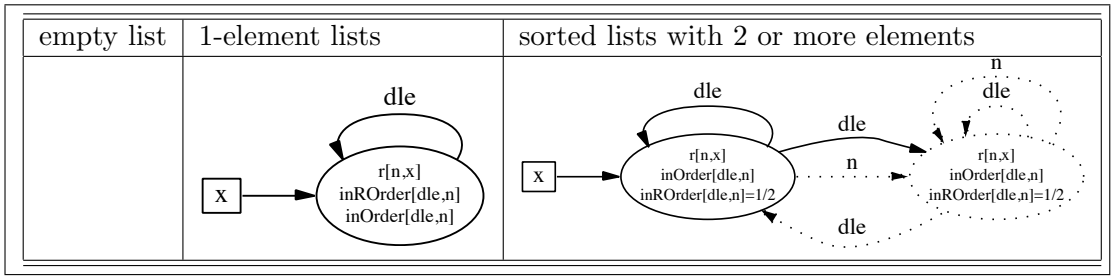


Figure 27: The structures that arise at l_2.

If the formula evaluates to 1, each node reachable from `x` is in non-decreasing order.

However, at this point, the reader may smell a rat: A "sorting" procedure that always returns `NULL` will satisfy Formula (8) at the exit point! Thus, Formula (8) is only part of the specification of the post-condition of a correct sorting procedure. A second property required of a correct sorting procedure (as well as of many other procedures that manipulate sorted linked lists) is that the output list must be a permutation of the input list.

We can establish that the permutation property holds for the output of `insert_sort` by extending the program-analysis specification with another predicate, $orig[n, x](v)$, whose value is set at the entry point to record the elements that are reachable from `x` there. In each statement, the predicate-update formula used for $orig[n, x]$ is $orig'[n, x](v) = orig[n, x](v)$. In other words, $orig[n, x]$ serves as an indelible mark on the elements initially reachable from `x`. At the end of the procedure, we then need to check that the following formula evaluates to 1:

$$\forall v : orig[n, x](v) \Leftrightarrow r[n, x](v). \tag{9}$$

If the formula does evaluate to 1, then the elements reachable from `x` after the procedure executes are exactly the same as those reachable at the beginning of the procedure, and consequently the procedure performs a permutation.

Example 4.2 Figure 27 shows the three 3-valued structures that arise at the end of `insert_sort`, given the structures shown in Figure 26 as the input structures. The structures in Figure 27 describe all possible stores in which variable `x` points to an acyclic, unshared, *sorted* linked list. In all

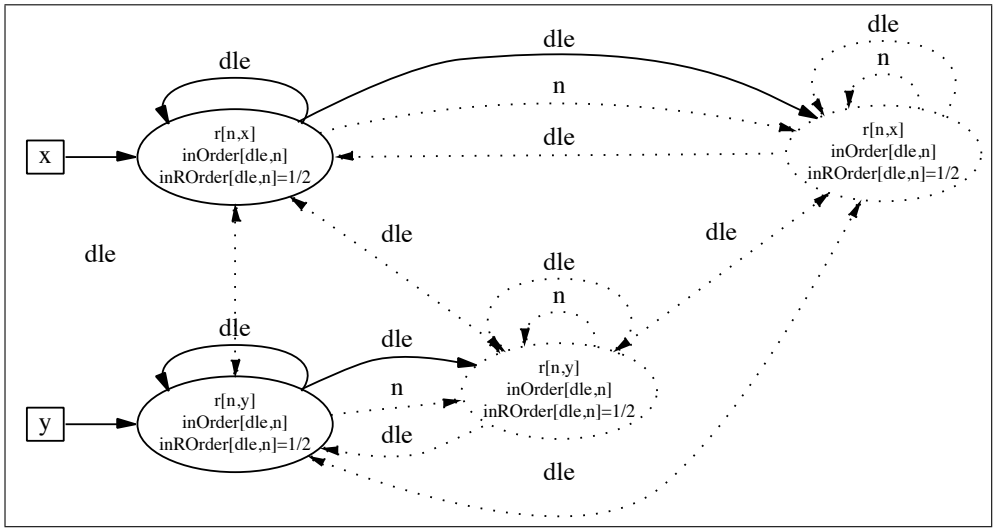


Figure 28: A structure that arises at l_4.

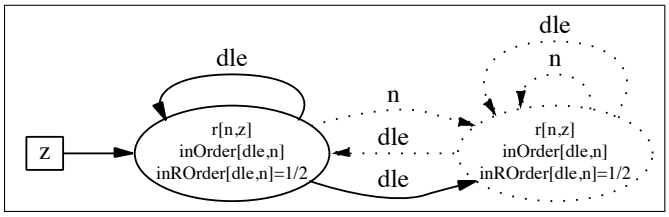


Figure 29: A structure that arises at l_5.

three structures, Formulas (8) and (9) both evaluate to 1.[4] Consequently, `insert_sort` is guaranteed to work correctly on all acceptable inputs.

Example 4.3 Figure 28 shows one of the structures that can arise at program point l_4 of main (see Figure 19). In Figure 28, the substructure consisting of the `x`-box together with the upper two nodes represents one sorted list of length 2 or more; the `y`-box and the lower two nodes represents a second sorted list of length 2 or more.

Figure 29 shows what is produced by the analysis when the structure shown in Figure 28 is supplied as the input structure in the analysis of

[4]Assuming, in the case of Formula (9), that instrumentation predicate $orig[n, x]$ was added to the analysis.

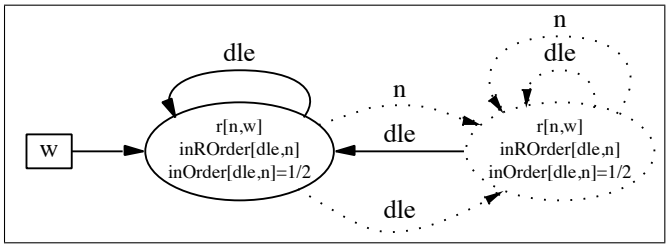


Figure 30: A structure that arises at l_6.

`merge`: The output structure represents the acyclic, unshared, *sorted* linked lists of length 2 or more. In other words, `merge` preserves sortedness.

Figure 30 shows what is produced when the structure shown in Figure 29 is supplied as the input structure in the analysis of `reverse`: The output structure represents the acyclic, unshared, linked lists of length 2 or more, *sorted in reverse order*.

Overall, the method described above is able to establish that at program point l_6 of main (Figure 19), program variable `w`—which is computed by reversing a list created by merging two sorted lists—always points to a list sorted in non-increasing order.

4.4 Mobile Ambients

The ambient calculus is a prototype web-language that allows processes (in the form of mobile ambients) to move inside a hierarchy of administrative domains (also in the form of mobile ambients); since the processes may continue to execute during their movement this notion of mobility extends that found in Java where only passive code in the form of applets may be moved. Mobile ambients were introduced in [CG]. The calculus is patterned after the π-calculus but focuses on named ambients and their movement rather than on channel-based communication; indeed, already the communication-free fragment of the calculus is very powerful (and in particular Turing complete). For a description of mobile ambients and 3-valued logic based analysis of them see [NNS00]. For the rest of the section, the reader is presumed to be farmiliar with [NNS00] and its results.

The program analyzed is a simulated router (see Figure 34). The packet p starts in the router r_1 and from there routed to router r_2. From r_2 it can be routed either back to r_1 or to r_3. Since processes may evolve when moving around it is hard to predict which ambients may turn up inside what other

```
/* amb_pred.tvp */
/*********************************************/
/************** Core Predicates **************/
foreach (z in AmbientNames) { %p z(v1) }
foreach (z in AmbientNames) {
    %p in[z](v1) // v1 is an in[z] capability
    %p out[z](v1) // v1 is an out[z] capability
    %p open[z](v1) // v1 is an open[z] capability
}
foreach (l in Labels) { %p l(v1) }

%p bang(v) // v is a replication operator (bang)
%p pa(v1, v2) function// v2 is the parent of v1
%p error()

/*************************************************************/
/*************** Instrumentation Predicates ******************/

foreach (z1 in AmbientNames) {
    foreach (z2 in AmbientNames) {
        %i inside[z1, z2]() = ∃v1, v2 : z1(v1) ∧ z2(v2) ∧ pa*(v1, v2)
    }
}
```

Figure 31: The predicates used in mobile-ambient analysis.

ambients. In this section we present an analysis that allows us to validate whether all executions satisfy properties like:

- Is there always exactly one copy of the ambient p?
- Is p always inside at most one of the ambients r_1, r_2 and r_3?

The analysis performed here is the analysis described in Section 3 with the exception that both Focus and Coerce and disabled. The predicates (see Figure 31) and transformers (see Figure 32) are defined according to the operational semantics of the replication operator (bang) and the different capabilities (open, out, in). The actions use the syntax of composite operators which allow to use logical operators on a set of formulae. For details see Appendix C. In case of ambients the TVS is very important as

/* amb_stat.tvp */
#define NB(v) $\vee/\{z(v) : z$ **in** AmbientNames$\}$
#define NBA(v) $(\forall v_1 : pa^*(v, v_1) \rightarrow \text{NB}(v_1))$
#define REDEX_PATH(v) $(\text{NB}(v) \wedge \text{NBA}(v) \wedge \neg error())$
%action in(m) { **%t** "in[" + m + "]"
%p $(pa(f_c, f_p) \wedge pa(f_p, f_{pp}) \wedge pa(f_s, f_{pp}) \wedge \text{REDEX\_PATH}(f_{pp}) \wedge$
$in[m](f_c) \wedge m(f_s) \wedge \text{NB}(f_p) \wedge \neg pa(f_c, f_s))$
{ $pa(v_1, v_2) = (v_1 = f_p \wedge v_2 = f_s) \vee (pa(v_1, f_c) \wedge v_2 = f_p) \vee$
$(pa(v_1, v_2) \wedge \neg(v_1 = f_p \wedge v_2 = f_{pp}) \wedge \neg(v_1 = f_c \wedge v_2 = f_p) \wedge \neg(v_2 = f_c))$
foreach (z **in**AmbientNames) {
$inside[z, m]() = inside[z, m]() \vee \exists v : z(v) \wedge pa^*(v, f_p)$
}
} **%retain** $v \neq f_c$
}
%action out(m) { **%t** "out[" + m + "]"
%p $(pa(f_c, f_p) \wedge pa(f_p, f_{pp}) \wedge pa(f_{pp}, f_{ppp}) \wedge \text{REDEX\_PATH}(f_{ppp}) \wedge$
$out[m](f_c) \wedge \text{NB}(f_p) \wedge m(f_{pp}))$
{ $pa(v_1, v_2) = (v_1 = f_p \wedge v_2 = f_{ppp}) \vee (pa(v_1, f_c) \wedge v_2 = f_p) \vee$
$(pa(v_1, v_2) \wedge \neg(v_1 = f_p \wedge v_2 = f_{pp}) \wedge \neg(v_1 = f_c \wedge v_2 = f_p) \wedge \neg(v_2 = f_c))$
foreach (z **in** AmbientNames) {
$inside[z, m]() = inside[z, m]() \wedge$
/* z is descendent of f_p and f_{pp} has an ancestor m */
$((\exists v_1, v_2 : z(v_1) \wedge pa^*(v_1, f_p) \wedge m(v_2) \wedge pa^+(f_{pp}, v_2)) \vee$
/* z is a descendent of m not through f_p */
$(\exists v_1, v_2 : z(v_1) \wedge m(v_2) \wedge pa^*(v_1, v_2) \wedge \neg(pa^*(v_1, f_p) \wedge pa^*(f_p, v_2))))$
}
} **%retain** $v \neq f_c$
}
%action open(m) {
%t "open[" + m + "]"
%p $(pa(f_c, f_p) \wedge pa(f_s, f_p) \wedge \text{REDEX\_PATH}(f_p) \wedge open[m](f_c) \wedge m(f_s))$
{ $pa(v_1, v_2) = (pa(v_1, f_c) \wedge v_2 = f_p) \vee (pa(v_1, f_s) \wedge v_2 = f_p) \vee (pa(v_1, v_2) \wedge$
$\neg(v_1 = f_c \wedge v_2 = f_p) \wedge \neg(v_2 = f_c) \wedge \neg(v_1 = f_s \wedge v_2 = f_p) \wedge \neg(v_2 = f_s))$
foreach (z **in** AmbientNames) {
$inside[z, m]() = inside[z, m]() \wedge$
/* z is descendent of f_s and f_p has an ancestor m */
$((\exists v_1, v_2 : z(v_1) \wedge pa^*(v_1, f_s) \wedge m(v_2) \wedge pa^*(f_p, v_2)) \vee$
/* z is a descendent of m not through f_s */
$(\exists v_1, v_2 : z(v_1) \wedge m(v_2) \wedge pa^+(v_1, v_2) \wedge \neg(pa^*(v_1, f_s) \wedge pa^*(f_s, v_2))))$
}
} **%retain** $v \neq f_c \wedge v \neq f_s$
}
%action nothing() { }

Figure 32: The actions used in mobile-ambient analysis.

%action bang() { **%t** "!"
%p $(pa(f_c, f_p) \wedge \text{REDEX\_PATH}(f_p) \wedge bang(f_c))$
%new $pa^+(v, f_c)$
{ $pa(v_1, v_2) = pa(v_1, v_2) \vee (\exists v_1 1 : instance(v_1, v_1 1) \wedge$
$((pa(v_1 1, f_c) \wedge v_2 = f_p) \vee (\exists v_2 1 : instance(v_2, v_2 1) \wedge pa(v_1 1, v_2 1))))$
foreach (z **in** AmbientNames) {
$z(v) = z(v) \vee \exists v_1 : (instance(v, v_1) \wedge z(v_1))$
$in[z](v) = in[z](v) \vee \exists v_1 : (instance(v, v_1) \wedge in[z](v_1))$
$out[z](v) = out[z](v) \vee \exists v_1 : (instance(v, v_1) \wedge out[z](v_1))$
$open[z](v) = open[z](v) \vee \exists v_1 : (instance(v, v_1) \wedge open[z](v_1))$
}
}
}

Figure 33: The actions used in mobile-ambient analysis continued from Figure 32.

it defines most of the actual program. For the TVS of the router example see Figure 35.

The non-determinism needed in defining the semantics (the choice of which ambient preforms its capability) is simulated using preconditions. The preconditions hold for assignments that represent ambients that are allowed to preform. Thus, all the possible sequences of capability activation are tried.

The bang operator is simulated using the **%new** (new) operator which replicates all the nodes which satisfy the formula of being decendents of the bang operator. The capabilities that are spent during the action are discarded using the **%retain** (retain) operator.

Since an unlimited number of bangs can occur between each capability activation and the most conservative case is that bang is performed more than once between each activation we can save calculation time by creating two control flow nodes "all" and "cap". Bang is done on a self loop on the "all" and all the capabilities on edges between "all" and "cap". Control then returns to the "all" node using the "nothing" action which doesn't change the state. Since TVLA uses reverse post order in the iterative algorithm the capability edges will not by traversed until the bang operation is fully explored.

The precision of the algorithm is increased by maintaining more instru-

/********** Sets **********/
%s AmbientNames $\{p, r, r_1, r_2, r_3, top\}$
%s Labels $\{l_0, l_1, l_2, l_3, l_4, l_5, l_6, l_7, l_8, l_9, l_{10}, l_{11},$
$l_{12}, l_{13}, l_{14}, l_{15}, l_{16}, l_{17}, l_{18}, l_{19}, l_{20}, l_{21}, l_{22}\}$
#include "amb_pred.tvp" // see Figure 31
%%
#include "amb_stat.tvp" // see Figure 32

%action verify_unique() { **%t** "verify_unique"
%p $p(v_1) \wedge p(v_2) \wedge (v_1 \neq v_2)$
%message $1 \rightarrow$ "Unique breached!"
{ error() = 1 }
}

%action verify_position() { **%t** "verify_position"
%p $p(v_1) \wedge \vee/\{z(v_2)\colon\ z$ **in** $\{r_1, r_2, r_3\}\} \wedge \vee/\{z(v_3)\colon z$ **in** $\{r_1, r_2, r_3\}\} \wedge (v_2 \neq v_3) \wedge$
$pa^*(v_1, v_2) \wedge pa^*(v_1, v_3)$
%message $1 \rightarrow$ "Position breached!"
{ error() = 1 }
}
%%
all bang() all
all in(r_1) cap
all in(r_2) cap
all in(r_3) cap
all in(p) cap
all out(r_1) cap
all out(r_2) cap
all open(r) cap
cap nothing() all
all verify_position() bug
all verify_unique() bug

Figure 34: The TVP for the router mobile-ambient analysis.

$$
\begin{array}{l}
\%\mathrm{n} = \{u_0, u_1, u_2, u_3, u_4, u_5, u_6, u_7, u_8, u_9, u_{10}, u_{11}, \\
\quad u_{12}, u_{13}, u_{14}, u_{15}, u_{16}, u_{17}, u_{18}, u_{19}, u_{20}, u_{21}, u_{22}\} \\
\%\mathrm{p} = \{ \\
\quad top = \{u_0\} \\
\quad r_1 = \{u_2\} \\
\quad r_2 = \{u_3\} \\
\quad r_3 = \{u_4\} \\
\quad p = \{u_1\} \\
\quad r = \{u_{11}, u_{12}, u_{13}\} \\
\quad bang = \{u_6, u_7, u_8, u_9\} \\
\quad in[p] = \{u_{14}, u_{15}, u_{16}\} \\
\quad in[r_1] = \{u_5, u_{22}\} \\
\quad in[r_2] = \{u_{20}\} \\
\quad in[r_3] = \{u_{21}\} \\
\quad out[r_1] = \{u_{17}\} \\
\quad out[r_2] = \{u_{18}, u_{19}\} \\
\quad open[r] = \{u_{10}\} \\
\quad pa = \{u_1\texttt{->}u_0, u_2\texttt{->}u_0, u_3\texttt{->}u_0, u_4\texttt{->}u_0, u_5\texttt{->}u_1, u_6\texttt{->}u_1, u_7\texttt{->}u_2, u_8\texttt{->}u_3, \\
\qquad u_9\texttt{->}u_3, u_{10}\texttt{->}u_6, u_{11}\texttt{->}u_7, u_{12}\texttt{->}u_8, u_{13}\texttt{->}u_9, u_{14}\texttt{->}u_{11}, u_{15}\texttt{->}u_{12}, \\
\qquad u_{16}\texttt{->}u_{13}, u_{17}\texttt{->}u_{14}, u_{18}\texttt{->}u_{15}, u_{19}\texttt{->}u_{16}, u_{20}\texttt{->}u_{17}, u_{21}\texttt{->}u_{18}, u_{22}\texttt{->}u_{19}\} \\
\quad l_0 = \{u_0\} \\
\quad \ldots \\
\quad l_{22} = \{u_{22}\} \\
\quad inside[p, top] = 1 \\
\quad inside[r, top] = 1 \\
\quad inside[r_1, top] = 1 \\
\quad inside[r_2, top] = 1 \\
\quad inside[r_3, top] = 1 \\
\quad inside[r, r_1] = 1 \\
\quad inside[r, r_2] = 1 \\
\}
\end{array}
$$

Figure 35: The TVS for the router mobile-ambient analysis.

mentation information on the forests represeted by a given 3-valued structure. The instrumentation defined here is inside[m,n] capturing which ambient is inside which ambient. The defining formula for the instrumentation is:

$$inside[m,n] = \exists v_1, v_2 : n(v_1) \wedge n(v_2) \wedge pa^+(v_1, v_2)$$

In a nondeterministic program the number of 3-valued structures arising in the analysis may be too big! (the worst case is exponential in the program size). In the routing example we got more than 10000 structures. The analysis can still be feasible by merging many strutures with the same property into a single struture (potentially loosing more precision). In the routing example merging all the structures with the same inside[m, n] leads to 72 structures.

5 The Coerce Algorithm

5.1 Constraints

Constraints are useful in improving the precision of the analysis. Constraints (also called *consistency rules*) are global invariants in the program. For example, the constraint

$$\exists v_1 : (\neg is[n](v) \wedge n(v_1, v) \wedge v_1 \neq v_2) \triangleright \neg n(v_2, v)$$

states that if node u is not shared by `n`-fields ($is[n]^S(u) = 0$) and already has an incoming `n`-field, it cannot have another incoming `n`-field. Thus if there exist $u_1 \neq u_2$ such that $n^S(u_1, u) = n^S(u_2, u) = 1$ the structure doesn't represent any valid concrete structure and can be discarded. Similarly, if there exist $u_1 \neq u_2$ such that $n(u_1, u) = 1$ and $n(u_2, u) = 1/2$ we can conservatively modify n so that $n(u_2, u) = 0$. For example, in Figure 8 we use this constraint on structure S_{o1} to remove the indefinite self loop on u, and on S_{o2} to remove the indefinite self loop on $u.1$ and the indefinite edge from $u.0$ to $u.1$.

In general a constraint is given in the form of $\varphi_1 \triangleright \varphi_2$, where φ_2 is a literal; φ_1 is called the *body* of the constraint and φ_2 is called the *head* of the constraint. The free variables of φ_1 and φ_2 must match. Such a constraint is *satisfied* on a structure S if for every assignment for which the body (φ_1) is evaluated to 1 (not just 1/2), the head must also evaluate to 1, denoted by $S \models \varphi_1 \triangleright \varphi_2$. Otherwise we say that the constraint is *breached* denoted by $S \not\models \varphi_1 \triangleright \varphi_2$.

If a constraint is breached by an assignment, the system tries to satisfy the constraint by changing 1/2 values into 1 or 0 as needed. If it is not possible to satisfy the constraint, the structure is invalid and is discarded. Formally, we define Coerce as follows,

Definition 5.1 *For a structure S and a set of constraints XF, the operation $Coerce(S, XF)$ returns the maximal S' such that $S' \sqsubseteq S$, $U^{S'} = U^S$, and S' does not breach any of the constraints in XF. Coerce is undefined if no such S' exists.*

The effects of Coerce on the precision of the analysis are noticeable. For example, Figure 8 shows how the imprecision resulting from the Focus operation are fixed. The constraints used for this Coerce operation are the ones automatically generated from the functional properties of the program variables and fields and from the instrumentation predicates used (see Section 5.2).

In this section we gradually explain the Coerce algorithm implemented in TVLA. For empirical results of how the new algorithms affect the performance of the system see Appendix A.

5.2 Automatic Generation of Constraints

While our system allows arbitrary constraints it is instructive to explain how constraints are automatically generated from functional properties and instrumentation predicates.

Functional Properties: The user can specify in TVP the functional properties of the predicates, and each of these functional properties can be naturally defined by a 2-valued formula. The system generates from this formula a set of constraints that maintain the truth of the formula. For example, for the declarations in Figure 7, the system generates the constraints shown in Table 5. The constraints generated include the defining formula of the property and the implied constraints of this formula as generated by the *closure* operator. The closure operator is defined as follows:

Definition 5.2 *For an extended Horn clause φ, we define the* **closure** *of φ, denoted by $closure(\varphi)$, to be the following set of constraints:*

$$closure(\varphi) \stackrel{\text{def}}{=} \left\{ \exists v_1, v_2, \ldots, v_n : \bigwedge_{i=1, i\neq j}^{m} \varphi_i^{1-B_i} \rhd \varphi_j^{B_j} \;\middle|\; \begin{array}{l} 1 \leq j \leq m, \\ v_k \in freeVars(\varphi), \\ v_k \notin freeVars(\varphi_j) \end{array} \right\} \tag{10}$$

For a formula φ that is not an extended Horn clause, $closure(\varphi) = \phi$.

Functional property	Constraint
%p $x(v_1)$ **unique**	(1) $x(v_1) \wedge x(v_2) \triangleright v_1 = v_2$ (2) $\exists v_1 : x(v_1) \wedge v_1 \neq v_2 \triangleright \neg x(v_2)$
%p $t(v_1)$ **unique**	(3) $t(v_1) \wedge t(v_2) \triangleright v_1 = v_2$ (4) $\exists v_1 : t(v_1) \wedge v_1 \neq v_2 \triangleright \neg t(v_2)$
%p $y(v_1)$ **unique**	(5) $y(v_1) \wedge y(v_2) \triangleright v_1 = v_2$ (6) $\exists v_1 : y(v_1) \wedge v_1 \neq v_2 \triangleright \neg y(v_2)$
%p $n(v_1, v_2)$ **function**	(7) $\exists v : n(v, v_1) \wedge n(v, v_2) \triangleright v_1 = v_2$ (8) $\exists v_1 : n(v, v_1) \wedge v_1 \neq v_2 \triangleright \neg n(v, v_2)$

Table 5: The constraints generated from functional properties shown in Figure 7.

For a complete list of all the functional properties supported and their corresponding constraints see Appendix C.

For an instrumentation predicate p defined by formula φ the following constraints are generated: (i) $\varphi \triangleright p$, (ii) $\neg\varphi \triangleright \neg p$, and (iii) the closure of $\varphi \rightarrow p$, and $\neg\varphi \rightarrow \neg p$. For example, all the constraints generated for instrumentation predicates of the reverse example are given in Table 6.

5.3 Order of Constraints

In most of the analyses we tried, the bottleneck of the analysis is the Coerce operation. This made Coerce the main candidate for optimization. One of the major problems in the Coerce algorithm given in [SRW99] is that after each change in the structure all the constraints need to be reevaluated to find if the last change caused a breach in one of them. Therefore, an expensive iterative fixed point algorithm is used. The TVLA system uses a sophisticated ordering of constraints that enables us in many cases to compute Coerce in one pass over the constraints. Specifically, we use the concept of depenencies between constraints. A constraint c_2 *depends* on constraint c_1 if while Coerce repairs c_1 on a structure that initially satisfies c_2 may cause a breach of c_2. Formally,

Definition 5.3 *If there exists a structure S such that $S \models c_2$ and Coerce$(S, c_1) \not\models c_2$ then we say that c_2* ***depends on*** *c_1.*

If we can find an ordering of constraints in which every constraint is evaluated before its depenents then after a single pass over all the constraints we

Instrumentation	**Generated Constraint**
%i $is[n](v) = \exists v_1, v_2 :$ $(n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2)$	(9) $\exists v_1, v_2 : (n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2) \triangleright$ $is[n](v)$ (10) $\neg\exists v_1, v_2 : (n(v_1, v) \wedge n(v_2, v) \wedge v_1 \neq v_2) \triangleright$ $\neg is[n](v)$ (11) $\exists v :(\neg is[n](v) \wedge n(v_1, v) \wedge n(v_2, v)) \triangleright$ $v_1 = v_2$ (12) $\exists v_2 :(\neg is[n](v) \wedge n(v_2, v) \wedge v_1 \neq v_2) \triangleright$ $\neg n(v_1, v)$ (13) $\exists v_1 :(\neg is[n](v) \wedge n(v_1, v) \wedge v_1 \neq v_2) \triangleright$ $\neg n(v_2, v)$
%i $r[n, x](v) =$ $\exists v_1 : (x(v_1) \wedge n^*(v_1, v))$	(14) $\exists v_1 : (x(v_1) \wedge n^*(v_1, v)) \triangleright r[n, x](v)$ (15) $\neg\exists v_1 : (x(v_1) \wedge n^*(v_1, v)) \triangleright \neg r[n, x](v)$
%i $r[n, y](v) =$ $\exists v_1 : (y(v_1) \wedge n^*(v_1, v))$	(16) $\exists v_1 : (y(v_1) \wedge n^*(v_1, v)) \triangleright r[n, y](v)$ (17) $\neg\exists v_1 : (y(v_1) \wedge n^*(v_1, v)) \triangleright \neg r[n, y](v)$
%i $r[n, t](v) =$ $\exists v_1 : (t(v_1) \wedge n^*(v_1, v))$	(18) $\exists v_1 : (t(v_1) \wedge n^*(v_1, v)) \triangleright r[n, t](v)$ (19) $\neg\exists v_1 : (t(v_1) \wedge n^*(v_1, v)) \triangleright \neg r[n, t](v)$
%i $c[n](v) = n^+(v, v)$	(20) $n^+(v, v) \triangleright c[n](v)$ (21) $\neg n^+(v, v) \triangleright \neg c[n](v)$

Table 6: The constraints generated from the instrumentation predicates shown in Figure 7.

are guaranteed that all the constraints hold (unless the structure is discarded in the process).

We define an initial dependcy graph between constraints where there is a directed edge from $c_1 \equiv l_1 \triangleright p^B(v_1, \ldots, v_k)$ to $c_2 \equiv l_2 \triangleright r_2$ if the literal $p^B(v'_1, \ldots, v'_k)$ appears in the DNF form of l_2. It is not hard to see that if there is no edge from c_1 to c_2 then c_2 does not depend on c_1.

The initial graph for the reverse example is given in Figure 36. As seen in the graph, the closure operator used for generating constraints from instrumentation predicates creates some problematic constraints that remain as loops in the dependecy graph (e.g. $10 \rightarrow 11 \rightarrow 10$). The truth is that they can be ignored, as stated in the next Lemma.

Lemma 5.4 *For a structure S such that $S \models \exists \ldots \varphi_1 \wedge \ldots \wedge \varphi_n \triangleright \varphi$, $coerce(S, \exists \ldots \varphi_1 \wedge \ldots \wedge \varphi_{i-1} \wedge \varphi_{i+1} \wedge \ldots \wedge \varphi_n \wedge \neg\varphi \triangleright \neg\varphi_i) \models \exists \ldots \varphi_1 \wedge \ldots \wedge \varphi_n \triangleright \varphi$*

Proof: Lets mark $S' = coerce(S, \exists \ldots \varphi_1 \wedge \ldots \wedge \varphi_{i-1} \wedge \varphi_{i+1} \wedge \ldots \wedge \varphi_n \wedge \neg\varphi \triangleright \neg\varphi_i$

By contradiction, assume that $S' \not\models \exists \ldots \varphi_1 \wedge \ldots \wedge \varphi_n \triangleright \varphi$. Since $S \models \exists \ldots \varphi_1 \wedge \ldots \wedge \varphi_n \triangleright \varphi$ and coerce can only make 1/2 into 1 or 0, there exists Z s.t. $[\![\varphi_1 \wedge \ldots \wedge \varphi_n]\!]^{S'}(Z) = 1$ and $[\![\varphi]\!]^{S'}(Z) = 0$. Thus, $[\![\varphi_1 \wedge \ldots \wedge \varphi_n \wedge \neg\varphi]\!]^{S'}(Z) = 1$. In other words

$$[\![\exists \ldots \varphi_1 \wedge \ldots \wedge \varphi_{i-1} \wedge \varphi_{i+1} \wedge \ldots \wedge \varphi_n \wedge \neg\varphi]\!]^{S'}(Z) = 1 \text{ and } [\![\neg\varphi_i]\!]^{S'}(Z) = 0$$

thus

$$S' \not\models \exists \ldots \varphi_1 \wedge \ldots \wedge \varphi_{i-1} \wedge \varphi_{i+1} \wedge \ldots \wedge \varphi_n \wedge \neg\varphi \triangleright \neg\varphi_i$$

in contradiction to the definition of coerce.

This resolves most of the loops in the dependecy graph. In the case of the singly linked list programs analyzed (reverse, merge, insert, etc.) there are no loops at all.

We now calculate a quasi-toplogical order of the graph and calculate Coerce using an iterative algorithm using this order. The initial work-set contains all the constraints. If a constraint causes a modification of the structure all its dependets are added to the work-set.

5.4 Memoizing Transitive Closures

Despite the fact that in many cases we can avoid reevaluating the defining formula of instrumentation predicates containing transitive closure in the update formula, transitive closure is still needed in (generated) constraints

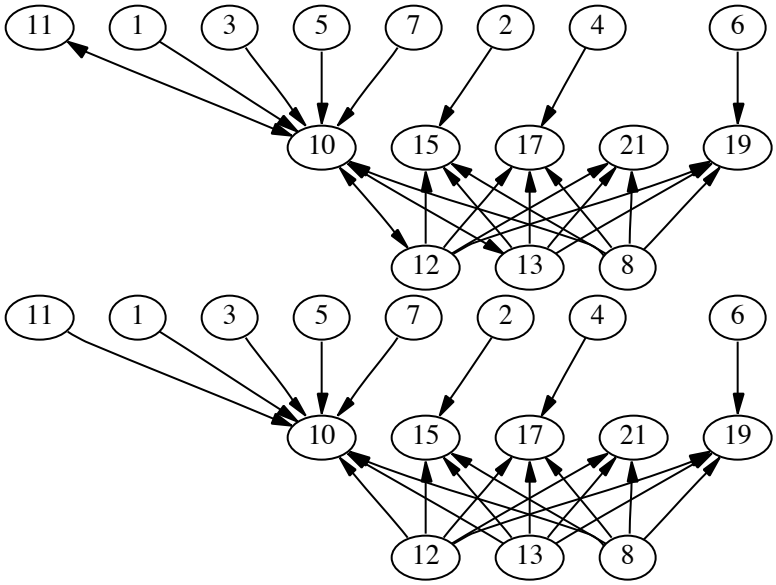


Figure 36: The dependency graph created for the reverse program, before and after the application of Lemma 5.4.

that can improve precision. Transitive closure is the most expensive operation in the formula evaluation. We use the Kleene semiring algorithm for calculating the transitive closure, which is $O(n^3)$ ([Ull89]).

For example, the "reachability from a program variable" instrumentation creates many constraints containing exactly the same transitive closure subformula. To save time in recalculating the transitive closure, each different transitive closure subformula is precalculated and replaced with a binary predicate in all the places it appears. Every time the predicates in the underlying subformula change, the transitive closure is recalculated and the relevant constraints are reevaluated.

5.5 Incremental Evaluation

Some actions have a very local effect on the structure. Some predicates' values remain unmodified after the action. If the constraint's body contains only atomic formulae that were unmodified since the last time the constraint was evaluated, there is no need to reevaluate the constraint. We maintain a dirty flag for each predicate, resetting it after each Coerce and setting it every time a predicate's value is modified. The initial work-set of the

iterative algorithms can now contain only the constraints that contain a dirty predicate or all the constraints if the universe of the structure was modified.

5.6 Incremental Formula Evaluation

When trying to evaluate a formula, a standard method is generating all the possible assignments into free variables of the formula, and evaluating the formula for each assignment.

Many of the useful constraints we use can be expressed as conjunctions of literals. If this is the case we use an efficient method taken from the deductive database world to evaluate the assignments that breach a constraint.

The Coerce operation needs all the assigments that evaluate the body of the constraint to true and the head to false or unknown. We order the literals (of both the body and the head) in the following order: unary, negated unary, binary, negated binary, equality, inequality. We then evaluate the formula incrementally in order, in each step we compute the relevant assignments by joining the previous assignments with the assignments that satisfy the current literal (or do not satisfy the head).

6 The Focus Algorithm

The focus operation is crucial for precise analysis. However, it can generate a large number of structures. The algorithm presented here uses the functional properties of the predicates to reduce the number of generated structures.

Some focus formulae might generate an infinite number of structures. For example, focusing on the formula $\forall v_1 : \neg n(v, v_1)$ (which is true for the last element of a singly linked list) on the structure shown in Figure 4 leads to an infinite number of structures. Our analysis conservatively identifies the cases where a specific formula might Focus a specific structure into an infinite number of structures and issues an error message at Focus time. A warning is given in compile time if a formula can potentially Focus a structure into an infinite number of structures.

This section presents a new algorithm for the Focus operation applicable on a general formula. The algorithm is more efficient in terms of number of structures generated than the one given in [SRW00] in cases where it is comparable. However, it still may create many unnecessary structures in the process.

The definition of the Focus operation is as follows:

```
Focus(S: structure, φ: formula) : Set of structures
conj_1 ∨ ... ∨ conj_n := DNF(QuatifierFree(TC_Eliminate(φ)))
XS_0 := {S}
for i in 1..n do
    XS_i := FocusConjunction(XS_{i-1}, conj_i)
od
return XS_n
```

Figure 37: The general Focus algorithm. DNF, QuatifierFree and TC_Eliminate are defined in Section 6.1 and FocusConjunction is defined in Figure 38.

Definition 6.1 *Given a set of formulae F,* **a focus operation for** *F is an operation that converts every structure S into a set of structures XS such that*

- *XS and S represent the same concrete structures,*
- *Every formula $\varphi \in F$ has a definite value in each of the structures in XS on every possible assignment.*

The Focus algorithm is given in Figure 37. The algorithm starts by normalizing the general formula into a set of conjunctions of literals (see Section 6.1). Each conjunction is analyzed incrementally to find the sources of indefinite assignments (Section 6.2). For each such indefinite assignment, the structure is replaced with a set of structures embedding the same set of concrete structures on which that assignment is no longer indefinite (Section 6.3).

The algorithm can benefit from the use of functional properties (Section 6.4) both in terms of the number of structures generated and in terms of "focusable" structures (structures that do not result in an infinite number of structures).

The actual implementation of the algorithm needs to overcome some inefficiencies in algorithm as presented here. A sketch of the solution to these inefficiencies is given in Section 6.5.

6.1 Normalizing Focus Formulae

Focus starts by normalizing the focus formula into a set of conjunctions of literals. Focusing on the resulting conjunctions yields structures in which the original formula is definite.

```
FocusConjunction(XS_0: Set of structs, φ_1 ∧ ... ∧ φ_n: conj. of literals) : Set of structs
for i in 1...n do
    XS_i := FocusLiteral(XS_{i-1}, φ_1 ∧ ... ∧ φ_{i-1}, φ_i)
od
return XS_n
```

Figure 38: Focus on a conjunction of literals. FocusLiteral is defined in Figure 39.

```
FocusLiteral(XS: Set of structures, φ : formula, φ' : literal) : Set of structures
AnswerSet := φ
while XS ≠ φ do
        select and remove S from XS
        if exists Z such that [[φ ∧ φ']]^S(Z) = 1/2 do
            XS := XS ∪ FocusAssignment(S, φ', Z)
        else
            AnswerSet := AnswerSet ∪ {S}
        fi
od
returnAnswerSet
```

Figure 39: Focus on a literal within a conjunction of literals. FocusAssignment is defined in Figure 40.

```
function FocusAssignment(S: structure, φ: Literal, Z assignment) : Set of structures
sw3Ditch φ
   c3Dase φ ≡ 1/2
    error(Illegal focus formula)
   c3Dase φ ≡ v1 = v2
    error(Focusing on equality of a summary node Z(v1)
          may create infinite number of structures)
   c3Dase φ ≡ p(v1, v2, ..., vk)
    Let ui = Z(vi)
    S0 := S[p^S(u1, u2, ..., uk) ↦ 0]
    S1 := S[p^S(u1, u2, ..., uk) ↦ 1]
    XS := {S0, S1}
    i3Df there exists i, s.t., sm^S(ui) = 1/2 then
     i3Df there exists j ≠ i, s.t., sm^S(uj) = 1/2 then
      error(Focusing on a formula φ with two summary nodes ui and uj
            may create an infinite number of structures)
     fi
     let u.0 and u.1 be individuals not in U^S
     S' := Expand(S, ui, u.0, u.1)
     S'' := S'[p^S'(u1, ..., ui−1, u.0, ui+1, ..., uk) ↦ 0, p^S'(u1, ..., ui−1, u.1, ui+1, ..., uk) ↦ 1]
     XS := XS ∪{S''}
    fi
    return XS
   c3Dase φ ≡ ¬φ'
    return FocusAssignment(S, φ', Z)
esac
```

Figure 40: Returns a set of structures that can be embedded into S such that φ evaluates to a definite value for the assignment Z.

First, *TC_Eliminate* removes all transitive closure operators assuring that the free variables of the sub-formulae remain unique. After focusing on the resulting formula every assignment evaluates the TC sub-formula to a definite value thus the transitive closure of the sub-formula must also evaluate to a definite value. Thus we can safely focus on the formula after *TC_Eliminate* instead of the original formula (As a result of this normalization we may conservatively report an error even when the original formula would have evaluated to a definite value).

Next, *QuatifierFree* removes all quantifiers. Focusing on resulting formula guarantees that all the assignments evaluate the formula to a definite value, thus the original formula which has quantifiers on of some of the variables must also evaluate to a definite value. Therefore, we can safely focus on the formula after *QuatifierFree* instead of the original formula.

Finally *DNF* is used to convert the quantifier free formula to DNF. Instead of focusing on the disjunction we decompose it into a set containing its composing conjunctions. Since focusing on all the conjunctions in the set means that for every assignment into the formula's free variables, all the conjunctions will be evaluated to a definite value, their disjunction must also evaluate to a definite value.

For example, consider the formula $\exists v_1 : x(v_1) \wedge n(v_1,\ v)$ which is used in interpreting the instruction `x = x->n` of the reverse example. The formula does not contain transitive closure and it is already in DNF. All that remains to normalize this formula is to remove the existential quantifier $\exists v_1$. The resulting formula is $x(v_1) \wedge n(v_1,\ v)$, which is already a conjunction. Thus in Figure 37, $conj_1 = x(v_1) \wedge n(v_1,\ v)$.

6.2 Focusing on Conjunctions of Literals

The normalization done to the focus formula in the previous section resulted in conjunctions of literals. In this section we show how to Focus on such formulae. The algorithm for FocusConjunction is given in Figure 38.

The algorithm for FocusConjunction is defined inductively. After step i, the set of structures XS_i is focused in respect to $\varphi_1 \wedge \ldots \wedge \varphi_i$ and embeds the same structures as XS_{i-1}. The initial set is XS_0 and the final result is XS_n. At each step FocusLiteral is called with a set of structures and two formulae. The precondition is that all the structures in the set are already focused for the first formula and should be focused for the conjunction of the two formulae.

FocusLiteral is a work-set algorithm which finds all the assignments in which the formula $\varphi \wedge \varphi'$ is indefinite (since the structures are focused in

FocusConjunction		FocusLiteral				
i	XS_i	φ	φ'	iteration	XS	AnswerSet
0	$\{S_{in}\}$					
		1	$x(v_1)$	0	$\{S_{in}\}$	ϕ
				1	ϕ	$\{S_{in}\}$
1	$\{S_{in}\}$					
		$x(v_1)$	$n(v_1, v)$	0	$\{S_{in}\}$	ϕ
				1	$\{S_{f0}, S_{f1}, S_{f2}\}$	ϕ
				2	$\{S_{f1}, S_{f2}\}$	$\{S_{f0}\}$
				3	$\{S_{f2}\}$	$\{S_{f0}, S_{f1}\}$
				4	ϕ	$\{S_{f0}, S_{f1}, S_{f2}\}$
2	$\{S_{f0}, S_{f1}, S_{f2}\}$					

Table 7: FocusConjunction on S_{in} given in Figure 8.

terms of φ it can only be because of φ'). For each indefinite assignment, FocusAssignment is called to handle it. If no such assignment exists then by definition, the structure is focused in terms of this formula and it is moved to the AnswerSet.

For example, consider the Focus operation shown in Figure 8. The initial input to the FocusConjunction algorithm is the set $\{S_{in}\}$ and the normalized Focus formula is $x(v_1) \wedge n(v_1,\ v)$ as shown in the previous section. Thus XS_0 $= \{S_{in}\}$ and for the first call to FocusLiteral we have XS $= \{S_{in}\}$, $\varphi \equiv 1$ and $\varphi' \equiv x(v_1)$. Since for every assignment Z we have $[\![x(v_1)]\!]^{S_{in}}(Z) \neq 1/2$, S_{in} is added to the AnswerSet of FocusLiteral and returned to FocusConjunction to become XS_1. Thus $XS_1 = \{S_{in}\}$ and for the second call to FocusLiteral we have XS $= \{S_{in}\}$, $\varphi \equiv x(v_1)$, and $\varphi' \equiv n(v_1,\ v)$. The assignment $Z = \{v_1 \mapsto u_0, v_2 \mapsto u\}$ is the only assignment for which $[\![x(v_1) \wedge n(v_1, v)]\!]^{S_{in}}(Z) = 1/2$, we call FocusAssignment with S_{in}, $n(v_1, v)$, and Z and the result is the three structures S_{f0}, S_{f1}, and S_{f2} (see Section /refSe:FocusAssignment). Thus of the second iteration of the loop $XS = \{S_{f0}, S_{f1}, S_{f2}\}$. For each of these structures no assignment evaluates the formula to 1/2 thus all three of them are added to the AnswerSet and returned to FocusConjunction to become XS_2. Since this is the last step the final result is $\{S_{f0}, S_{f1}, S_{f2}\}$.

6.3 Focusing on Literals

FocusAssignment is called with a structure S, a literal φ and an assignment Z such that $[\![\varphi]\!]^S(Z) = 1/2$. A case analysis is performed. If the formula

is a negation we call the function recursively without the negation (since negation does not change the definiteness of a formula). If the formula is 1/2, it is invalid since the Focus formula should be 2-valued. If the formula is an equality then the only possible reason for FocusAssignment to be called is if the assignment maps both variables into a the same summary node. In this case no Focus is possible and an error message is given.

The common case is that the formula is a k-ary predicate. Using the assignment we compute the relevant tuple of the predicate whose value is 1/2. In this case we need to check that no more than one of nodes in the tuple is a summary node. If two or more of the nodes in the tuple are summary nodes then the Focus will result in an infinite structure and an error message is given. If none of the nodes in the tuple is a summary node then there are two cases for the value of this tuple in structures embedded into the given structure - 0 and 1. We thus generate two structures one with the value of the tuple forced to 0 and one with the value of the tuple forced to 1. If one of the nodes in the tuple is a summary node then there is a third case - a structure where some of the nodes embedded to the summary node result in a tuple whose value is 0 and some of the nodes result in a tuple whose value is 1. For this kind of structures we create a third structure where the summary node is expanded into two summary nodes (the values of all the predicates for these new summary nodes are equal to the value of the original summary node) for one of the summary nodes the value of the tuple is 0 and for the other 1.

For example, as described in the last section FocusAssignment is called once for $n(v_1, v)$, with $S = S_i n$ and $Z = \{v_1 \mapsto u_0, v \mapsto u\}$. Since u is a summary node and u_0 isn't we generate three structures, S_{f0}, S_{f1}, and S_{f2} as shown in Figure 8.

6.4 Using Functional Properties in Focus

The functional properties of the predicates give us an interesting insight into the structures that Focus should generate. For example, if a unary predicate is declared **unique** then Focus should never generate a structure where that predicate holds for two different nodes because that structure does not represent any valid concrete structure and will be discarded by Coerce. This optimization can be very important in terms of the number of structures created For example, focusing on a **unique** unary predicate that evaluates to 1/2 on m nodes creates $m + 1$ structures instead of $> 2^m$ structures.

Another use of the functional properties is to turn summary nodes into

non-summary nodes whenever possible. For example, if an unary predicate is declared **unique** and holds for a summary node then the summary node can be safely turned into a non-summary node (since the predicate can not be true for more than one node). This feature is important since as we saw in Section 6.3 an infinite number of structures is created when a binary predicate is focused on a self loop of a summary node, thus, turning summary nodes into non-summary nodes in the process enables us to focus on formulae that would otherwise create an infinite number structures.

For example, look at Figure 8. The binary predicate n is a **function**. Taking advantage of this fact in the focus, we can turn u into a non-summary node in the S_{f1} and $u.1$ into a non-summary node in S_{f2}.

The most important functional properties in terms of focus are **unique**, **function**, and **invfunction** since they enable us to perform both optimizations. When taking functional properties into account the order in which we focus the literals of the conjunction is important since we want to turn summary nodes into non-summary nodes as soon as possible. A possible good order is to focus first on all the predicates that have **unique**, **function**, or **invfunction** functional properties and only then on the rest of the predicates.

6.5 The Actual Implementation

The actual implementation of the Focus operation has another problem to consider. The algorithm as specified here required recomputing all the assigments that evaluate the formula to 1/2 for each structure and after every change. This can be very inefficient. The actual implementation keeps track for each structure the partial assignments that satisfy the sub-formula and in each step only considers assignments that extend them. The algorithm has to deal with another problem when computing $\cup$ of sets of structures. These structures are not necessarily bounded structures. Computing $\cup$ requires comparing structures to check that they are not already in the set. Without bounded structures we are forced to use general isomorphism between graphs as comparison which is exponential. Without comparing the structures, the algorithm would create the same structure many times, which is very inefficient.

The source of the problem is in focusing on a conjunction. The algorithm focuses in each step on a literal for each assignment that satisfies the preceding sub-formula. For assignments that differ only in variables not used in the current literal the same structures are created.

The solution to the problem is similar to an optimization done in de-

ductive databases. The assignments are projected into the relevant set of variables and the literal is focused only with assignments that differ in variables used in the given literal. The assignments are not recalculated at every iteration, they are accumulated as the focus proceeds, and projected in each step onto the set of variables still needed in future iterations.

The algorithm as currently implemented in TVLA has several drawbacks. One, it only supports nullary, unary and binary predicates. Support for n-ary predicates is not available but can be added. Two, certain formulae can cause cases were the algorithm returns structures on which the formula still evaluates to 1/2. The result is ofcourse still conservative (i.e., the set of embedded concrete structures is still the same).

7 Active Nodes

TVLA also supports dynamic creation and deletion of individuals from the structure. Such changes are necessary in order to model statements such as `malloc` and `free`, which can affect the actual existence of concrete nodes. TVLA's support for dynamic changes of individuals is very general and goes well beyond manipulation needed for shape analysis. Furthermore, it also supports very compact representation of structures which is comparable to [SRW98], thereby solving a major open problem in [SRW99].

Conceptually, a TVLA user can assume that the number of individuals is infinite. However, in every structure the number of *active* individuals is always finite. Operations such as `malloc` and `free` are modeled in the TVP by modifying the active property of individuals.

The active property of individuals is maintained by a designated unary predicate ac, where $ac^S(u) = 1$ indicates that the individual u is active in the structure S. In shape analysis, $ac^S(u)$ can be one of the following,

- 1 — indicating that u represents at least one concrete node in all the concrete structures represented by this structure.
- 1/2 — indicating that u may not be present in some structures (i.e., it is possible that u does not represent a concrete node in one or more of the concrete structures represented by this structure).
- 0 — indicating that u does not represent any concrete node in any of the concrete structures represented by this structure, and can thus be discarded by the implementation.

The rest of this section is organized as follows: We start with a generalized definition of embedding in Section 7.1. We then show what modifications should be done to formula evaluation (Section 7.2), Coerce (Section 7.3), and Focus (Section 7.4). Section 7.5 defines the mechanisms that support creating new nodes (*new*) and removing unneeded nodes (*retain*). We conclude with a description of single structure analysis (Section 7.6).

7.1 Generalized Embedding

Definition 7.1 *Let S and S' be two structures. Let $f\colon U^S \to U^{S'}$. We say that f* **embeds** *S in S' (denoted by $S \sqsubseteq^f S'$) if (i) for every predicate p (including sm and ac) of arity k and all $u_1, \ldots, u_k \in U^S$,*

$$p^S(u_1, \ldots, u_k) \sqsubseteq p^{S'}(f(u_1), \ldots, f(u_k)) \tag{11}$$

and (ii) for all $u' \in U^{S'}$

$$(|\{u \mid f(u) = u'\}| > 1) \sqsubseteq sm^{S'}(u') \tag{12}$$

and (iii) for all $u' \in U^{S'}$ having $ac^{S'}(u') = 1$, there exists $u \in U^S$ such that $f(u) = u'$.

We say that S **can be embedded in** *S' (denoted by $S \sqsubseteq S'$) if there exists a function f such that $S \sqsubseteq^f S'$.*

If all the individuals are active then this definition is equivalent to Definition 2.3, since the third requirement implies that f is surjective.

An equivalent embedding theorem exists for the generalized version of embedding, i.e., formulae with definite values in S' agree on these values with every S embedded into S'.

Theorem 7.2 [Generalized Embedding Theorem]. *Let S, S' be two structures, and let $f\colon U^S \to U^{S'}$ be a function such that $S \sqsubseteq^f S'$. Then, for every formula φ and complete assignment Z for φ, $[\![\varphi]\!]^S(Z) \sqsubseteq [\![\varphi]\!]^{S'}(f \circ Z)$.*
Proof: Appears in Appendix B.

7.2 Formula Evaluation

Formula evaluation is modified slightly in to handle maybe active nodes. Let Z be an assignment such that $S, Z \models \varphi$. If Z contains a node which is maybe active, it is possible that in some concrete structures it does not exist, thus, the formula must evaluate to unknown for that assignment. The new definition of the meaning of a formula is given below.

Definition 7.3 *The* **meaning** *of a formula φ, denoted by $[\![\varphi]\!]^S(Z)$, yields a truth value in $\{0, 1, 1/2\}$. The meaning of φ is defined inductively as follows:*

Atomic *For a logical literal $\boldsymbol{l} \in \{\mathbf{0}, \mathbf{1}, \mathbf{1/2}\}$, $[\![\boldsymbol{l}]\!]^S(Z) = l$ (where $l \in \{0, 1, 1/2\}$).*

For an atomic formula $p(v_1, \ldots, v_k)$,

$$[\![p(v_1, \ldots, v_k)]\!]^S(Z) = p^S(Z(v_1), \ldots, Z(v_k))$$

For an atomic formula $(v_1 = v_2)$,

$$[\![v_1 = v_2]\!]^S(Z) = \begin{cases} 0 & Z(v_1) \neq Z(v_2) \\ 1 & Z(v_1) = Z(v_2) \text{ and } sm^S(Z(v_1)) = 0 \\ 1/2 & \text{otherwise} \end{cases}$$

Logical Connectives *For logical formulae φ_1 and φ_2*

$$\begin{aligned} [\![\varphi_1 \wedge \varphi_2]\!]^S(Z) &= \min([\![\varphi_1]\!]^S(Z), [\![\varphi_2]\!]^S(Z)) \\ [\![\varphi_1 \vee \varphi_2]\!]^S(Z) &= \max([\![\varphi_1]\!]^S(Z), [\![\varphi_2]\!]^S(Z)) \\ [\![\neg\varphi_1]\!]^S(Z) &= 1 - [\![\varphi_1]\!]^S(Z) \end{aligned}$$

Quantifiers *If φ is a logical formula,*

$$\begin{aligned} [\![\forall v_1 : \varphi]\!]^S(Z) &= \min_{u \in U^S} \max(1 - ac^S(u), [\![\varphi_1]\!]^S(Z[v_1 \mapsto u])) \\ [\![\exists v_1 : \varphi]\!]^S(Z) &= \max_{u \in U^S} \min(ac^S(u), [\![\varphi_1]\!]^S(Z[v_1 \mapsto u])) \end{aligned}$$

Transitive Closure *For $(TC\ v_1, v_2 : \varphi)(v_3, v_4)$,*

$$[\![(TC\ v_1, v_2 : \varphi)(v_3, v_4)]\!]^S(Z) = \max_{\substack{n \geq 1, u_1, \ldots, u_{n+1} \in U, \\ Z(v_3) = u_1, Z(v_4) = u_{n+1}}} \min(\min_{i=1}^{n} [\![\varphi]\!]^S(Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}]), \min_{i=2}^{n} ac^S(u_i))$$

As for transitive closure, the requirment is that every path that includes a node which is maybe active should give 1/2. Since we use the semi-ring algorithm for calculating the transitive closure, the only change needed is that the update formula changes from

$$n^*(v_1, v_2) = n^*(v_1, v_2) \vee (n^*(v_1, v) \wedge n^*(v, v_2))$$

to

$$n^*(v_1, v_2) = n^*(v_1, v_2) \vee (ac(v) \wedge n^*(v_1, v) \wedge n^*(v, v_2))$$

7.3 Coerce

The effect on Coerce is of course more severe. If the assignment that breaches a constraint contains maybe active nodes we must ignore that breach since the body evaluates to unknown.

7.4 Focus

The Focus operation is also affected by the addition of maybe active nodes. The definition of Focus requires the formula to evaluate to unknown if the assignment contains maybe active nodes, this means that the focus operation must make sure that all the assignments that contain maybe active nodes and thus should be evaluated to unknown are focused in a sense that each maybe active node might either exist or not exist. In the algorithm, every time a new variable is added to the assignment and is bound to a maybe active node, the structure is replaced with two structures, one in which the node exists and is active and another were the node does not exist.

7.5 Actions

We now describe how the concept of active nodes is used to model operations that create new nodes or remove unneeded nodes.

7.5.1 New

Some actions require the creation of new nodes (e.g., in response of a `malloc` statement). Two forms of **%new** are supported: (i) create a single new node, and (ii) given an unary formula, duplicate all the nodes that potentially satisfy the formula. In both cases a new predicate *isNew* is introduced to the duration of this action and set to 1 for each node created in this action. In the case (ii), a binary predicate called *instance* is introduced connecting each duplicated node with its matching new node. If the duplicated node was a summary node, the new node is a summary node too.

For a **%new** with an unary formula $\varphi(v_1)$ we evaluate

$$ac^S(v) = ac^S(v) \vee \exists v_1 : \varphi(v_1) \wedge instance(v, v_1)$$

to determine which nodes should be active after the new. Notice that the formula $\varphi(v_1)$ might evaluate to 1/2 in which case $ac^S(v) = 1/2$ because some concrete structures represented by S may not satisfy $\varphi(v_1)$ and thus v should not be in them.

7.5.2 Retain

Some actions require the deletion of nodes (e.g., in response to a free command). We use **%retain** with a unary formula φ specifying which nodes should be retained after the action. Nodes that do not potentially satisfy the formula are removed along with the associated values in the predicates.

For a **%retain** with an unary formula we evaluate

$$ac^S(v) = ac^S(v) \wedge \varphi(v)$$

to determine which nodes should be active after the new. Notice that the formula $\varphi(v)$ might evaluate to 1/2 in which case $ac^S(v) = 1/2$ because some concrete structures represented by S may not satisfy $\varphi(v)$ and thus v should still exist in them.

7.6 Single Structure

Single structure analysis is a well known idiom in the program analysis community. It is used a for more efficient (in terms of time and space) but less precise analysis. The support for maybe active nodes gives us a way to represent all the structures in at a CFG node in a single structure. The algorithm needed to complete the picture is it that of joining a new structure with an existing structure resulting in a structure that represents all the concrete structures that either of these structures represent.

The Join algorithm works when both structures are blurred using the same set of abstraction predicates. The nodes are renamed to their canonic names before applying the Join. Join is given in Figure 41.

7.6.1 Using Nullary Predicates to Improve Precision

The main problem with single structure analysis is the loss of precision. Fortunately, in most of the programs analyzed we are able to reconcrete structure precision by modifying the join operation slightly. The join no longer creates a single structure, but at most 3^k structures were k is the number of nullary predicates in the analysis. Two structures that differ in the values of their nullary predicates are not merged (this is similar to the concept of abstraction in unary predicates).

For shape analysis, we add the instrumentation

$$nn[x]() = \exists(v) : x(v)$$

which states that the program variable `x` is not null. The results are promising since most of the singly list functions and even the bubble sort with

Join(S_1, S_2: structure) : structure
$U^S = U^{S_1} \cup U^{S_2}$
foreach $u \in U^S - (U^{S_1} \cap U^{S_2})$ **do**
    $ac^S(u) = 1/2$
od
foreach $u \in U^{S_1} \cap U^{S_2}$ **do**
    $ac^S(u) = ac^{S_1}(u) \sqcup ac^{S_2}(u)$
od
foreach $p \in P$ of arity 0
    $p^S() = p^{S_1}() \sqcup p^{S_2}()$
od
foreach $p \in P - \{ac\}$ of arity 1 **do**
    foreach $u \in U^{S_1} - U^{S_2}$ **do**
        $p^S(u) = p^{S_1}(u)$
    od
    foreach $u \in U^{S_2} - U^{S_1}$ **do**
        $p^S(u) = p^{S_2}(u)$
    od
    foreach $u \in U^{S_1} \cap U^{S_2}$ **do**
        $p^S(u) = p^{S_1}(u) \sqcup p^{S_2}(u)$
    od
od
foreach $p \in P$ of arity 2 **do**
    foreach $u_1 \in U^{S_1} - U^{S_2}, u_2 \in U^{S_1}$ **do**
        $p^S(u_1, u_2) = p^{S_1}(u_1, u_2)$
        $p^S(u_2, u_1) = p^{S_1}(u_2, u_1)$
    od
    foreach $u_1 \in U^{S_2} - U^{S_1}, u_2 \in U^{S_2}$ **do**
        $p^S(u_1, u_2) = p^{S_2}(u_1, u_2)$
        $p^S(u_2, u_1) = p^{S_2}(u_2, u_1)$
    od
    foreach $u_1, u_2 \in U^{S_1} \cap U^{S_2}$ **do**
        $p^S(u_1, u_2) = p^{S_1}(u_1, u_2) \sqcup p^{S_2}(u_1, u_2)$
        $p^S(u_2, u_1) = p^{S_1}(u_2, u_1) \sqcup p^{S_2}(u_2, u_1)$
    od
od
return S

Figure 41: Joining two structures.

value swap function are analyzed accurately. For timing information for the single graph analysis see Appendix A.

8 Conclusion

The method of three-valued logic based program analysis can handle a wider class of problems than shape analysis. We have successfully analyzed Mobile Ambients even though it is a completely different paradigm. We can also show partial correctness of algorithms such as the sorting programs.

However, it is clear that some analyses go beyond the scope of TVLA and it is not obvious whether TVLA can or should be extended to support them. Specifically, the concrete semantics must be expressible using first order logic with transitive closure, in particular no arithmetic is supported.

Program analysis algorithms are hard to design, prove correct and implement. The concept of three-valued logic based analysis greatly simplifies the problem since it allows us to work with the concrete operational semantics instead of the abstract semantics. The use of three-valued logic guarantees that the transition to the abstract semantics is sound. TVLA has two major contributions to the simplification of problem. First, it provides a platform on which one can easily try new algorithms and observe the results. Second, the constraints generated from the instrumentation predicates are a very strong tool in assuring the correctness of the analysis. The instrumentation predicates are updated separately from the core predicates and any discrepancy between them causes a constraint breach which is reported by the system. We often found out that this exposes bugs in the concrete semantics. When a core or instrumentation predicate is updated in a way which is inconsistent with its functional properties or with the defining formula of some instrumentation predicate, a constraint breach happens.

A common principle in program analysis is that there is a trade-off between the time of analysis and its precision. In case of three-valued logic based analysis, this is not always true. A more precise analysis creates less unneeded structures and thus runs faster. A good example for this is the merge function (see Section 4.1) where adding the reachability information drastically reduces both the space and the time needed for the analysis.

The use of instrumentation predicates is a very good tool in improving precision, and have a very low cost. If a class of programs has an invariant that is true for most of them and can improve precision if the invariant holds but it doesn't necessarily hold in all the, using an instrumentation predicate to track that invariant allows us to use the power of the invariant without

limiting ourselves to programs that were that invariant holds. For example, we use cyclicity instrumentation to update the reachability information. If a singly linked list is acyclic updating the reachability information can be done more precisely. The use of the cyclicity instrumentation allows us to take advantage of this property without limiting the analysis to acyclic lists.

The adaptation of three-valued logic based analysis to single structure can in many cases help solve the space problems found in the analysis. An interesting insight is that not all the structures should be merged. A mechanism similar to abstraction should exist to allow differentiating between structures. The mechanism introduced here of using nullary predicates is shown to be sufficient for analyzing singly linked accurately.

The system was implemented in Java which is an Object-Oriented imperative language. The use of strong libraries such as the Collections library enables the development of complex data structure manipulation without using a more high level language such as ML. A prototype of the system was also written using a deductive database system (CORAL see [RSSS93]). However, since the operations needed are not exactly within the scope on the current deductive databases, the results were unsatisfactory. The analysis was not feasible even for a few loop iterations of the reverse example.

8.1 The Essence of Instrumentation

Our experience indicates that instrumentation predicates are essential to achieving efficient and useful analyses. First, they are helpful in debugging the operational semantics. The instrumentation predicates are updated separately from the core predicates, and any discrepancy between them is reported by the system. Our experience indicates that in many cases this reveals bugs in the operational semantics.

The conventional wisdom in static analysis is that there is a trade-off between the time of analysis and its precision (i.e., that a more precise analysis is more expensive). In case of 3-valued-logic-based analysis, this is not always true. Often it happens that an analysis that uses more instrumentation predicates creates fewer unneeded structures, and thus runs faster. A good example of this is the merge function (see Section 4.1) where adding the reachability information drastically reduces both the space and the time needed for the analysis.

In general, the introduction of instrumentation predicates is a very good tool for improving precision, and has a very low cost. If a property holds for many but not all nodes of the structures that arise in a program, then we can use an instrumentation predicate to track at which program points

and for which nodes the property holds. This allows us to use the implications of the property without limiting ourselves to programs where the property holds . For example, we use cyclicity instrumentation to update the reachability information. If a singly linked list is acyclic, updating the reachability information can be done more precisely. The use of cyclicity instrumentation allows us to take advantage of this property without limiting the analysis to programs in which lists are always acyclic. Of course, in some programs, such as `rotate`, where cyclicity is temporarily introduced, we may lose precision when evaluating formulae in 3-valued logic. This is in line with the inherent complexity of these problems. For example, updating reachability in general directed graphs is a difficult problem.

Formally, instrumentation predicates allow us to narrow the set of 2-valued structures represented by a 3-valued structure, and thereby avoid making overly conservative assumptions in the abstract interpretation of a statement. For example, the structure shown in Figure 4 represents an acyclic singly linked list, which means that all of the list elements represented by u are not shared. Thus, $is[n]^{S_4}(u) = 0$. The same holds for u_0. Without the sharing information, the structure might also represent 2-valued structures in which the linked list ends with a cycle back to itself.

For unary instrumentation predicates, we can fine-tune the precision of an analysis by varying the collection of predicates used as abstraction predicates. The more abstraction predicates used, the finer the distinctions that are made, which leads to a more precise analysis. For example, the fact that is is an abstraction predicate allow us to distinguish between shared and unshared list elements in programs such as the `swap` function, where a list element is temporarily shared. Of course, this may also increase the cost of the analysis.

8.2 Comparison to Related Work

TVLA is based on the theoretical framework introduced in [SRW99], [SRW00], and [NNS00]. The thesis introduces new algorithms for Coerce and Focus and extends the framework to handle single structure analysis. The thesis also introduces the concept of functional properties of predicates and their use in automatic constraint generation. The Coerce algorithm was optimized to avoid unnecessary recomputations by using lazy evaluation, imposing an order of constraint evaluation and using relational database query optimization techniques (see [Ull89]) to evaluate formulae. The Focus algorithm was generalized to handle an arbitrary formula. This was crucial to support the formulae used for analyzing sorting programs. In addition, the Focus algo-

rithm in TVLA was also optimized to take advantage of functional properties of the predicates.

The worst-case space of the analysis was improved from doubly exponential to singly exponential by means of the option in which all the structures with the same nullary predicate values are merged together. Thus, the number of potential structures becomes independent of the number of nodes in the structure. Interestingly, in most of the cases analyzed to date the analysis remains rather precise. However, in some cases it actually increases the space needed for the analysis due to decreased precision.

For an elaborate comparison of three-valued based analysis with other methods of program analysis see [SRW00]. TVLA is a framework for intra-procedural program analysis. There are frameworks for inter-procedural program analysis such as PAG ([AM95]), SHARLIT ([TH92]), OPTIMIX [Aßm98] and Vortex ([Cha99]). The main advantage of these systems over TVLA is the support for inter-procedural analysis and their scalability to larger programs. The main problem of these systems is that they force the user to work directly with the abstract semantics. This makes the process of developing new program analysis algorithms much harder especially for parametric analyses. In TVLA the user can specify the concrete operational semantics and the system takes care of the transition to the abstract semantics. It may be possible to integrate TVLA into one of these systems to benefit from both approaches.

BANE ([Aik99]) is a constraint solver based on set constraints. The constraints used in TVLA are first order logic constraints which are much stronger. Specifically, the number of constraints in BANE is bounded by the size of the program. In TVLA since the constraints can contain quantifiers and even transitive closure, the number of BANE like constraints is not bounded by the size of the program. On the other hand, BANE is flow-insensitive and usually much more space and time efficient than TVLA.

8.3 Further Work

The system is very useful in the analysis of small programs. However, there are many issues that should be solved before the analysis can scale to larger programs. Most of these are theoretical and not implementation issues.

The design of instrumentation predicates remain the hardest part in developing a good program analysis algorithm. Tools and methodologies to assist in the development of such predicates are needed. The system does take a step in the right direction by automatically generating constraints from the defining formula of the instrumentation and the functional prop-

erties of these predicate thus helps in maintaining the correctness of the update formulae for these predicates.

The choice of which of the predicates should be abstraction predicates is very important for the space/precision trade-off. We lack a good methodology for selecting these predicates.

The programs analyzed can have multiple types in them. Types can be simulated by using unary predicates. However, this approach creates problems in the automatic generation of constraints, and the complicates the definition of instrumentation predicates. A better approach may be to use multi-sorted logics to define the different types. The user will supply for each node in the input its type and every time a node is generated by the new command, a type will be associated with it.

The algorithm proposed for Focus cannot handle equality, although it should be possible to define an algorithm that can. The Focus algorithm handling of transitive closure also leaves something to be desired. For example, a focus on a binary predicate considers the whole structure instead of the path between the two given nodes, which can create more graphs than necessary.

The update formulae and preconditions are formulae that can have several quantified or free variables. The naive algorithms is exponential in the number of quantifications since all the possible assignments of free variables are generated before the formula is evaluated. Notice however that this number is usually independent of the size of the analyzed program, at least not in the cases we tried. Techniques for optimizing the formula evaluation be taken for the database world and adapted to Kleene logic.

The relationship between the structures generated during the analysis can be of interest especially in the case of flow-insensitive analysis. We could create a structure derivation graph and use it to answer temporal logic like queries. For example, in the case of Mobile Ambients, we could answer queries such as does the packet p always reside in the router r after it entered it.

The current analysis engine only supports intra-procedural analysis. The analysis of complex programs requires the extension of the system to support inter-procedural analysis. The problem of shape analysis in the context of procedures still lacks a good solution. The known methods such as call-string has major space problems and cannot handle recursion in reasonable precision.

Focus and Coerce are two semantic reductions used in the analysis. We want to find a way to unite the two mechanisms. This can be useful for example, in limiting the number of structures created by Focus that would

later be discarded in Coerce. Another possible unification is to describe the full operational semantics and the Focus operation as constraints and use a sophisticated constraint solver as the analysis engine.

The extensive use of logic in the system raises the question of how well could deductive database system handle the needed operation. A prototype of such a system was written, the initial results however, were not promising.

The major problem in terms of scalability of the system is the space needed for the analysis. We use some techniques to alleviate the problem but they are not enough. A possible solution to the problem may be to use Binary Decision Diagrams (BDDs) to represent multiple logical structures ([Bry92]). This will be particularly useful if we could find a way to apply the operations needed (Coerce, Focus, Blur) on a BDD without expending it back to the underlying logical structures (such methods are known to exist for formula evaluation). Another possible solution to the space program is the use of secondary storage for structures that are not participating in the current operation.

Acknowledgements

We are grateful for the helpful comments and contributions of N. Dor, M. Fähndrich, G. Laden, F. Nielson, H.R. Nielson, T. Reps, N. Rinetskey, R. Shaham, O. Shmueli, R. Wilhelm, and A. Yehudai.

References

[Aik99] A. Aiken. Bane (the berkeley analysis engine). Available at http://www.cs.berkeley.edu/Research/Aiken/bane.html, 1999.

[AM95] M. Alt and F. Martin. Generation of efficient interprocedural analyzers with PAG. In *SAS'95, Static Analysis Symposium*, LNCS 983, pages 33–50. Springer, September 1995.

[Aßm98] U. Aßmann. *Graph Grammar Handbook*, chapter OPTIMIX, A Tool for Rewriting and Optimizing Programs. Chapman-Hall, 1998.

[Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *Computing Surveys*, 24(3):293–318, September 1992. Available at http://www.cs.cmu.edu/∼bryant/pubdir/CMU-CS-92-160.ps.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Symp. on Princ. of Prog. Lang.*, pages 269–282, New York, NY, 1979. ACM Press.

[CG] L. Cardelli and A. D. Gordon.

[Cha99] C. Chambers. Vortex. "http://www.cs.washington.edu/research/projects/cecil/www/vortex. 1999.

[CWZ90] D.R. Chase, M. Wegman, and F. Zadeck. Analysis of pointers and structures. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 296–310, New York, NY, 1990. ACM Press.

[Deu94] A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 230–241, New York, NY, 1994. ACM Press.

[DRS00] N. Dor, M. Rodeh, and M. Sagiv. Checking cleanness in linked lists. In *SAS'00, Static Analysis Symposium*, 2000. Available at "http://www.math.tau.ac.il/∼ nurr".

[Eva96] D. Evans. Static detection of dynamic memory errors. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, 1996. Available at "http://larch-www.lcs.mit.edu:8001/∼ evs/pldi96-abstract.html".

[JJNS97] J.L. Jensen, M.E. Joergensen, N.Klarlund, and M.I. Schwartzbach. Automatic verification of pointer programs using monadic second-order logic. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, 1997.

[LA00] T. Lev-Ami. TVLA: A framework for Kleene based static analysis. Master's thesis, Tel-Aviv University, 2000. Available at http://www.math.tau.ac.il/∼tla.

[Muc99] S.S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan & Kaufmann, third edition, 1999.

[NNS00] F. Nielson, H.R. Nielson, and M. Sagiv. A kleene analysis of mobile ambients. In *Proceedings of the 2000 European Symposium On Programming*, March 2000. Available at "http://www.math.tau.ac.il/∼ sagiv".

[RSSS93] R. Ramakrishnan, P. Seshadri, D. Srivastava, and S. Sudarshan. Implementation of the CORAL deductive database system. In *Proceedings of the ACM SIGMOD 93 Conference*, pages 167–176, New York, NY, 1993. ACM Press.

[SRW98] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *Trans. on Prog. Lang. and Syst.*, 20(1):1–50, January 1998.

[SRW99] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Symp. on Princ. of Prog. Lang.*, 1999. Available at "http://www.cs.wisc.edu/wpis/papers/popl99.ps".

[SRW00] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. Tech. Rep. TR-1383, Comp. Sci. Dept., Univ. of Wisconsin, Madison, WI, March 2000. Submitted for publication. Available at "http://www.cs.wisc.edu/wpis/papers/tr1383.ps".

[TH92] S.W.K. Tjiang and J. Hennessy. Sharlit—a tool for building optimizers. In *SIGPLAN Conf. on Prog. Lang. Design and Impl.*, pages 82–93, June 1992.

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volume II: The New Technologies.* Comp. Sci. Press, Rockville, MD, 1989.

List of Figures

List of Tables

A Empirical

The timing information in this section is for a Linux system running on a PentiumII 400MHz. It is known that using JVM on Windows the system runs about 20% faster.

The system was used to analyze on a number of examples (see Section 4). The timing information for all the functions analyzed is given in Table 9, as are the results for the functions we analyzed in single structure mode. Notice, the number of structures created is total number of structures that were created and not discarded by the Coerce and not the number of structures kept in the CFG node.

The system utilizes several algorithms for improving the speed of the Coerce operation (see Section 5). The effect of each of the optimizations on the time Coerce takes is given in Table 8.

B Proof of the Generalized Embedding Theorem

Theorem 7.2 *Let S and S' be two structures, and let $f: U^S \to U^{S'}$ be a function such that $S \sqsubseteq^f S'$. Then, for every formula φ and complete assignment Z for φ, $[\![\varphi]\!]^S(Z) \sqsubseteq [\![\varphi]\!]^{S'}(f \circ Z)$.*

Function	All	No incremental formula evaluation	No transitive closure memoization	No constraint ordering	None
insert	1.714	2.672	2.769	2.956	6.305
reverse	0.625	0.861	1.116	1.085	2.392
getlast	0.359	0.479	0.811	0.515	1.5
search	0.29	0.357	0.706	0.458	1.177
create	0.247	0.294	0.314	0.299	0.321
delete	1.645	2.134	3.254	2.663	6.993
swap	0.305	0.378	0.481	0.396	0.737
null_deref	0.402	0.36	0.75	0.497	1.308
del_all	0.168	0.164	0.147	0.144	0.172
fumble	0.628	0.85	1.264	0.898	2.481
rotate	0.288	0.346	0.335	0.343	0.551
merge	5.503	8.47	11.909	11.209	35.801
insert sort	114.359	187.504	173.207	302.761	702.744

Table 8: Timing for Coerce with various levels of optimization

.

Proof: By the De Morgan laws it is sufficient to show the theorem for formulae involving $\wedge$, $\neg$, $\exists$, and TC. The proof is by structural induction on φ:
Basis: For atomic formula $p(v_1, v_2, \ldots, v_k)$, $u_1, u_2, \ldots, u_k \in U^S$, and $Z = [v_1 \mapsto u_1, v_2 \mapsto u_2, \ldots, v_k \mapsto u_k]$ we have

$$\begin{aligned}
&[\![p(v_1, v_2, \ldots, v_k)]\!]^S(Z) \\
&\quad = p^S(u_1, u_2, \ldots, u_k) && \text{(Definition 7.3)} \\
&\quad \sqsubseteq p^{S'}(f(u_1), f(u_2), \ldots, f(u_k)) && \text{(Definition 2.3)} \\
&\quad = [\![p(v_1, v_2, \ldots, v_k)]\!]^{S'}(f \circ Z) && \text{(Definition 7.3)}
\end{aligned}$$

Also, for $l \in \{0, 1, 1/2\}$, we have:

$$\begin{aligned}
&[\![\boldsymbol{l}]\!]^S(Z) \\
&\quad = l && \text{(Definition 7.3)} \\
&\quad \sqsubseteq l && \text{(Definition 2.1)} \\
&\quad = [\![\boldsymbol{l}]\!]^{S'}(f \circ Z) && \text{(Definition 7.3)}
\end{aligned}$$

Let us now show that

$$[\![v_1 = v_2]\!]^S(Z) \sqsubseteq [\![v_1 = v_2]\!]^{S'}(f \circ Z).$$

Function	**Multiple Structures**		**Single Structure**	
	Number of Structures	**Time (seconds)**	**Number of Structures**	**Time (seconds)**
insert	140	2.862	60	3.233
reverse	70	1.217	54	2.121
getlast	40	0.785	30	1.687
search	40	0.708	30	1.45
create	21	0.511	10	0.434
delete	145	2.739	92	6.073
swap	31	0.7	19	0.663
null_deref	48	0.752	37	1.511
del_all	11	0.42	9	0.446
fumble	81	1.406	56	2.135
rotate	25	0.629	17	0.92
merge	327	8.253	96	14.308
splice ([SRW99])	22	1.968		
splice (here)	22	1.144		
insert sort	3773	160.132		
bubble sort	3946	186.609		

Table 9: Time and number of structures for functions analyzed.

First, if $[\![v_1 = v_2]\!]^{S'}(f \circ Z) = 1/2$ then the theorem holds for $v_1 = v_2$, trivially. Second, if $[\![v_1 = v_2]\!]^{S'}(f \circ Z) = 1$ then by Definition 7.3, (i) $f(Z(v_1)) = f(Z(v_2))$ and (ii) $sm^{S'}(f(Z(v_1))) = 0$. Therefore, by Definition 2.3, $Z(v_1) = Z(v_2)$ and $sm^S(Z(v_1)) = 0$ both hold. Hence, by Definition 7.3, $[\![v_1 = v_2]\!]^S(Z) = 1$. Finally, suppose that $[\![v_1 = v_2]\!]^{S'}(f \circ Z) = 0$ holds. In this case, by Definition 7.3, $f(Z(v_1)) \neq f(Z(v_2))$. Therefore, $Z(v_1) \neq Z(v_2)$, and by Definition 7.3 $[\![v_1 = v_2]\!]^S(Z) = 0$.

Induction step: Suppose φ is a formula with free variables $v_1, v_2, \ldots v_k$. Let Z be a complete assignment for φ. If $[\![\varphi]\!]^{S'}(Z) = 1/2$, then the theorem holds trivially. Therefore assume that $[\![\varphi]\!]^{S'}(f \circ Z) \in \{0, 1\}$. We distinguish between the following cases:

Logical-and $\varphi \equiv \varphi_1 \wedge \varphi_2$. The proof splits into the following subcases:

Case 1: $[\![\varphi_1 \wedge \varphi_2]\!]^{S'}(f \circ Z) = 0$.
In this case, either $[\![\varphi_1]\!]^{S'}(f \circ Z) = 0$ or $[\![\varphi_2]\!]^{S'}(f \circ Z) = 0$. Without loss of generality assume that $[\![\varphi_1]\!]^{S'}(f \circ Z) = 0$. Then, by the induction hypothesis for φ_1, we conclude that $[\![\varphi_1]\!]^S(Z) = 0$. Therefore, by Definition 7.3, $[\![\varphi_1 \wedge \varphi_2]\!]^S(Z) = 0$.

Case 2: $[\![\varphi_1 \wedge \varphi_2]\!]^{S'}(f \circ Z) = 1$.
In this case, both $[\![\varphi_1]\!]^{S'}(f \circ Z) = 1$ and $[\![\varphi_2]\!]^{S'}(f \circ Z) = 1$. Then, by the induction hypothesis for φ_1 and φ_2, we conclude that $[\![\varphi_1]\!]^S(Z) = 1$ and
$[\![\varphi_2]\!]^S(Z) = 1$. Therefore, by Definition 7.3, $[\![\varphi_1 \wedge \varphi_2]\!]^S(Z) = 1$.

Logical-negation $\varphi \equiv \neg\varphi_1$. The proof splits into the following subcases:

Case 1: $[\![\neg\varphi_1]\!]^{S'}(f \circ Z) = 0$.
In this case, $[\![\varphi_1]\!]^{S'}(f \circ Z) = 1$.
Then, by the induction hypothesis for φ_1, we conclude that $[\![\varphi_1]\!]^S(Z) = 1$.
Therefore, by Definition 7.3, $[\![\neg\varphi_1]\!]^S(Z) = 0$.

Case 2: $[\![\neg\varphi_1]\!]^{S'}(f \circ Z) = 1$.
In this case, $[\![\varphi_1]\!]^{S'}(f \circ Z) = 0$.
Then, by the induction hypothesis for φ_1, we conclude that $[\![\varphi_1]\!]^S(Z) = 0$.
Therefore, by Definition 7.3, $[\![\neg\varphi_1]\!]^S(Z) = 1$.

Existential-Quantification $\varphi \equiv \exists v_0 : \varphi_1$. The proof splits into the following subcases:

Case 1: $[\![\exists v_1 : \varphi_1]\!]^{S'}(f \circ Z) = 0$.
In this case, for all $u \in U^S$, $[\![\varphi_1]\!]^{S'}((f \circ Z)[v_1 \mapsto f(u)]) = 0$. Then,

by the induction hypothesis for φ_1, we conclude that for all $u \in U^S$ $[\![\varphi_1]\!]^S(Z[v_1 \mapsto u]) = 0$. Therefore, by Definition 7.3, $[\![\exists v_1 : \varphi_1]\!]^S(Z) = 0$.

Case 2:$[\![\exists v_1 : \varphi_1]\!]^{S'}(f \circ Z) = 1$.
In this case, there exists a $u' \in U^{S'}$ such that $[\![\varphi_1]\!]^{S'}((f \circ Z)[v_1 \mapsto u']) = 1$. By Definition 7.3, $ac^{S'}(u') = 1$, thus by Definition 7.1, there exists a $u \in U^S$ such that $f(u) = u'$ and $[\![\varphi_1]\!]^{S'}((f \circ Z)[v_1 \mapsto f(u)]) = 1$. Then, by the induction hypothesis for φ_1, we conclude that $[\![\varphi_1]\!]^S(Z[v_1 \mapsto u]) = 1$, and by Definition 7.1 $ac^S(u) = 1$. Therefore, by Definition 7.3, $[\![\exists v_1 : \varphi_1]\!]^S(Z) = 1$.

Transitive Closure $\varphi \equiv (TC\ v_1, v_2 : \varphi_1)(v_3, v_4)$. The proof splits into the following subcases:

Case 1: $[\![(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)]\!]^{S'}(f \circ Z) = 1$.
Let $Z(v_3) = u_1$ and $Z(v_4) = u_{n+1}$ thus $(f \circ Z)(v_3) = u'_1$, and $(f \circ Z)(v_4) = u'_{n+1}$. By Definition 7.3, there exist $u'_1, u'_2, \ldots, u'_{n+1} \in U^{S'}$ such that for all $1 \leq i \leq n$, $[\![\varphi_1]\!]^{S'}((f \circ Z)[v_1 \mapsto u'_i, v_2 \mapsto u'_{i+1}]) = 1$. Also, for all $2 \leq i \leq n$, $ac^{S'}(u'_i) = 1$.

Thus, by Definition 7.3, there exist $u_2, \ldots, u_n \in U^S$ such that for all $2 \leq i \leq n$, $f(u_i) = u'_i$. Therefore, by the induction hypothesis, for all $1 \leq i \leq n$, $[\![\varphi_1]\!]^S(Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}]) = 1$. Also, by Definition 7.3, for all $2 \leq i \leq n$, $ac^S(u_i) = 1$. Hence, by Definition 7.3, $[\![(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)]\!]^S(Z) = 1$.

Case 2: $[\![(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)]\!]^{S'}(f \circ Z) = 0$.
We need to show that $[\![(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)]\!]^S(Z) = 0$. Assume on the contrary that $[\![(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)]\!]^{S'}(f \circ Z) = 0$, but $[\![(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)]\!]^S(Z) \neq 0$. Because $[\![(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)]\!]^S(Z) \neq 0$, by Definition 7.3 there exist $u_1, u_2, \ldots, u_{n+1} \in U^S$ such that $Z(v_3) = u_1$, $Z(v_4) = u_{n+1}$, and for all $1 \leq i \leq n$, $[\![\varphi_1]\!]^S(Z[v_1 \mapsto u_i, v_2 \mapsto u_{i+1}]) \neq 0$. Hence, by the induction hypothesis there exist $u'_1, u'_2, \ldots, u'_{n+1} \in U^{S'}$ such that $(f \circ Z)(v_3) = u'_1$, and $(f \circ Z)(v_4) = u'_{n+1}$ and for all $1 \leq i \leq n$, $[\![\varphi_1]\!]^{S'}((f \circ Z)[v_1 \mapsto u'_i, v_2 \mapsto u'_{i+1}]) \neq 0$. Therefore, by Definition 7.3, $[\![(TC\ v_1, v_2 : \varphi_1)(v_3, v_4)]\!]^{S'}(f \circ Z) \neq 0$, which is a contradiction.

```
%n = {u, u0}
%p = {
       sm = {u:1/2}
       n = {u ->u:1/2, u0 ->u:1/2}
       x = {u0:1}
       nn[x] = 1
       r[n,x] = {u:1, u0:1}
}
```

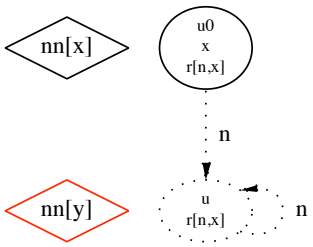


Figure 42: The TVS of an input structure for the reverse function analysis and its graphical representation.

C User's Manual

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

C.1 Graphical Representation

3-valued structures are displayed using graphical representation. For example an input structure for the reverse function is given in Figure 42.

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).

Shapes: A diamond represents a nullary predicate. The name of the predicate is written within the diamond. For example, in Figure 42, $nn[x]$ is true thus it is drawn as a solid black diamond and $nn[y]$ is false thus it is drawn as a red diamond.

An ellipsis represents a node (annotated with its name if **-significant** is used). If the ellipsis is dotted then the node is a summary node, and if it is green then the node is maybe active. Unary predicates are written within the ellipsis. If the value if different from 1 it is appended to predicate's name (i.e., $= 0$ or $= 1/2$).

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

```
/* Set of the program variables */
%s PVar {x, y, t}
/* Program variables definition */
foreach (z in PVar) { %p z(v1) unique box }
/* Selector definition */
%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)) }
/* Non-Null instrumentation */
foreach (z in PVar) { %i nn[z](v) = E(v1) z(v1) {1, 0, 1/2} }
/* Cyclicity instrumentation */
%i c[n](v) = n+(v, v)
```

Figure 43: The declarations part of the TVP program for the reverse function shown in Figure 1.

edges are replaced with a bidirected edge.

C.2 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 43, 44, and 45. The syntax of a TVP program is given in Figure 46. The syntax is in extended BNF when A $\bowtie$ B denotes a (possibly empty) sequence of A's separated by B's.

C.2.1 Predicates

TVLA supports nullary, unary and binary single-sorted predicates. The predicates are 3-valued, i.e. the value of the predicate at each tuple of nodes from the structure's universe is either true (1), false (0) or unknown (1/2).

The predicate name can be either <id> or <id>[<id>, ..., <id>], the latter is used when the predicate name is dependent on the names of other predicates (this is especially good for instrumentations). The predicate's arity is determined by the number of variables in parenthasis (currently the names of the variables are of no consequence). For a description of the proprties that can be used in predicate declaration see Table 10. The

```
/****************************** 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
      nn[x1]() = 0
      r[n, x1](v) = 0}
}
%action Copy_Variable_L(x1, x2) { %t x1 + " = " + x2
    %f { x2(v) }
    { x1(v) = x2(v)
      nn[x1]() = nn[x2]()
      r[n, x1](v) = r[n, 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)
      nn[x1]() = E(v1, v) x2(v1) & n(v1, v)
      r[n, x1](v) = r[n, x2](v) & (c[n](v) |  ! x2(v))}
}
%action Set_Next_Null_L(x1) { %t x1 + "->" + n + " = NULL"
    %f { x1(v) }
    { n(v1, v2) = n(v1, v2) &  ! x1(v1)
      is[n](v) = is[n](v) & ( ! (E(v1) x1(v1) & n(v1, v)) |
                              E(v1, v2) v1 ! = v2 & (n(v1, v) &  ! x1(v1)) &
                              (n(v2, v) &  ! x1(v2)))
      r[n, x1](v) = x1(v)
      foreach(z in PVar-{x1}) {
          r[n, z](v) =(c[n](v) & r[n, x1](v)?
                       z(v) | E(v1) z(v1) & TC(v1, v)(v3, v4)(n(v3, v4) &  ! x1(v3)) :
                       r[n, z](v) &  ! (r[n, x1](v) &  ! x1(v) & E(v1) r[n, z](v1) & x1(v1)))
      }
      c[n](v) = c[n](v) &  ! (E(v1) x1(v1) & c[n](v1) & r[n, x1](v))}
}
%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)
      is[n](v) = is[n](v) | E(v1) x2(v) & n(v1, v)
      foreach(z in PVar) {
          r[n, z](v) = r[n, z](v) | r[n, x2](v) & E(v1) r[n, z](v1) & x1(v1)
      }
      c[n](v) = c[n](v) | (r[n, x2](v) & E(v1) x1(v1) & r[n, x2](v1))}
}
```

Figure 44: The actions part of the TVP program for the reverse function shown in Figure 1.

```
/* The program's CFG and theeffect of its edges */
n_1 Set_Null_L(y) n_2              /* y = NULL; */
n_2 Is_Null_Var(x) exit            /* x == NULL */
n_2 Is_Not_Null_Var(x) n_3         /* x != NULL */
n_3 Copy_Variable_L(t, y) n_4      /* t = y; */
n_4 Copy_Variable_L(y, x) n_5      /* y = x; */
n_5 Get_Next_L(x, x) n_6           /* x = x->n; */
n_6 Set_Next_Null_L(y) n_7         /* y->n = NULL; */
n_7 Set_Next_L(y, t) n_8           /* y->n = t; */
n_8 Set_Null_L(t) n_2              /* t = NULL; */
```

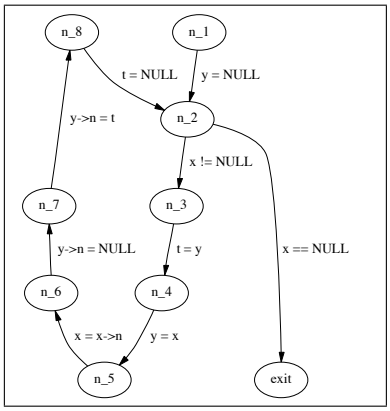


Figure 45: The CFG part of the TVP program for the reverse function shown in Figure 1 and its corresponding CFG.

<display> flag controls which values of the predicate are shown in the graph (in DOT). The default is {1, 1/2} specifying that true and unknown values are to be shown.

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 a instrumentation predicate the user has to attach its defining formula. The formula's free variables should match the variables given in the parenthasis (with the exception of precondition free variables explained later).

C.3 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 result TC relation. The formula $\varphi_{cond}?\varphi_{true} : \varphi_{false}$ is an if-then-else formula. If φ_{cond} evaluates to true the value of the formula is φ_{true}. If φ_{cond} evaluates to false the value of the formula is φ_{false}. If φ_{cond} evaluates to false the value of the formula is $\varphi_{true} \sqcup \varphi_{false}$.

C.3.1 Consistency Rules

Most of the needed consistency rules for an analysis are automatically generated from the functional properties of predicates (see Table 10) and from

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

Figure 46: The syntax of a TVP program.

Property	Arity	Meaning	Consistency Rule
unique	unary	true for atmost one node	p(v1) & p(v2) ==> v1 == v2 E(v1) p(v1) & v1 != v ==> !p(v)
function	binary	partial function	E(v) p(v, v1) & p(v, v2) ==> v1 == v2 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 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 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
box	unary	display p in a box	N/A

Table 10: 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 The value of the predicate for the assignment is unknown	The structure is invalid - discard. Coerce it to true.
negated predicate	The value of the predicate for the assignment is true The value of the predicate for the assignment is unknown	The structure is invalid - discard. Coerce it to false.
variable equality	The two variables are assigned to different nodes The variables are assigned to the same node and it is a summary node	The structure is invalid - discard. Coerce into a non summary node.
variable inequality	The two variables are assigned to the same node	The structure is invalid - discard

Table 11: Result of a consistency rule breach according to its head.

the instrumentation predicates defining formulae. Sometimes it is useful to write explicit consistency rules. The left hand side of the consistency rules (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 $\triangleright$. Notice 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 11.

C.3.2 Actions

The arguments of an action are predicate names that can be used in the following formulae and will be replaced with the actual arguments when the action is used (see Section C.4). The actions section of the reverse program is given in Figure 44.

*Title (**%t**)*: The title of the action, used when printing the action's

structures.

*Focus (**%f**)*: The focus formulae for this action. Applied before the precondition.

*Precondition (**%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.

*Report Messages (**%message**)*: Messages that are reported to the user if the formula given is potentially satisfied.

*New (**%new**)*: A mechanism for creating new nodes. 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 uknown for a certain node, the matching new node becomes maybe active.

Update Formulae: Update formulae dictate how predicates should be 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. The variables is parenthesis should number as the predicate's arity and should match the free variables of the formula. Notice that the update clauses are not comma separated.

*Retain (**%retain**)*: A mechanism for removing unwanted nodes. An unary formula must be supplied. Only nodes that potentially satisfy the formula are retained. If the formulae supplied evaluates to uknown for a certain node, it becomes maybe active instead of being removed.

C.4 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.

C.5 Usability

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

C.5.1 Comments

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

C.5.2 Preprocessing

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

C.5.3 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.

C.5.4 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 (<pred> **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.

```
<tvs> ::= <structure>*
<structure> ::= <universe> <predicates>
<universe> ::= %n = { <node>⋈, }
<predicates> ::= %p = { <predicate>* }
<predicate> ::= <pred> = <kleene> /* Nullary */
              | <pred> = { (<node> [<value>])⋈, } /* Unary */
              | <pred> = { (<leftnode>-><rightnode> [<value>])⋈, } /* Binary */
<node> ::= <id>
         | [ <node>⋈, ]
         | <node>.(0|1)
<value> ::= : <kleene>
```

Figure 47: The syntax of a TVS file.

C.5.5 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 natural 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).

C.6 TVS

The input structures for the analysis are described in a format call TVS (Three Valued Structure). For example, the TVS for input structure used in the analysis of the reverse function is given in Figure 42. A TVS file should end with the extension '.tvs'. The syntax of a TVS file is given in Figure 47.

The value of a predicate defaults to false if not specified differently 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 program. If a node (or a node pair) are specified without a value, the default of true (1) is taken. TVS supports the same commenting style as TVP.

C.7 Command Line Options

```
Usage: tvla <program name> <input file> [-ms <number>] [-mm <number>] [-d] [-post]
        [-join {all$|$ext}] [-b2] [-action [f][c]pu[c][b]] [-significant]
        [-single] [-log logfile] [-dump <number>] [-noautomatic] [-rotate]
```

Maximum number of structures: A complex analysis may take a very long time and especially in debug time may run idefinitly. 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).

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 structure 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 ireparable 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.

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.

Join locations: The default behaviour of the system is to preform a join (thus saving all the structures that reached the program location) only in every back edge in the control graph (approximatly once in every loop). Performing join at every program location is the most efficient in terms of the number of structures generated (use **-join all**). However, it is very space consuming since structures are saved at every program location. For a compromise between the two extremes use **-join ext** which preforms a join only at the beginning of each extended block (i.e., at every merge point in the control graph).

Blur: Two rules are implemented when determining which node should remain distinguished in the blur. The default is two nodes must be different in the value of at least one abstraction predicate. The second is two nodes must be different in a *definite* value of at least one abstraction predicate. Use the **-b2** flag to use the second rule.

Computing the effect of an action: Sometime you what to try and run the algorithms (Coerce, Focus, Precondition, Update, Blur) in a different order or quantity then the default one (Focus, 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.

Node names: It may be useful to track the nodes themselves from the original structures and during the different actions. Use the **-significant** to try and keep the names of the nodes significant. A materialization creates two nodes <node>.0 and <node>.1 according to the focused value of the predicate in question. A blur that causes merging of nodes names the new nodes [<node>,...,<node>]. However, the nodes allocated by the new operation use never before used node names in the form m<number>. In complex analyses using significant names may create very large names that create unreadable graphs.

Single Structure: The system supports single structure analysis using the **-single** flag. 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.

Log file: A log file name can be supplied. In this case the majority of the information written to the stderr is redirected to the log file.

Dumping intermidate results: Some of the analyses can take a very long time. To see intermidate results the **-dump** flag can be used. The number supplied is the number of structures generated between each dump of the current set of structures per program location.

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.

Landscape graphs: To draw the graphs from left to right instead of top to bottom use the **-rotate** flag.