# 3VMC User's Manual

draft version

Eran Yahav*

January 27, 2001

**Abstract**

3VMC is a model checker which uses abstraction based on 3-valued logic. $3VMC$ is an extension of the TVLA framework. This document is a user's manual for $3VMC$.

## 1 Introduction

$3VMC$ is a tool for analysis and verification of concurrent systems. In contrast to existing verification tools (e.g. [2]) $3VMC$ is geared towards verification of concurrent software with dynamic allocation of objects and threads. $3VMC$ provides a powerful abstraction mechanism by using 3-valued logic to represent multiple program configurations. $3VMC$ is an extension of the TVLA framework [3]. Theoretical background for $3VMC$ can be found in [5].

In this paper, we will show how to perform verification of the simple Java program given in Figure 1.

## 2 Three-Valued Configurations

A *program configuration* encodes a global state of a program which consists of (i) a global store, (ii) the program-location of every thread, and (iii) the status of locks and threads, e.g., if a thread is waiting for a lock. Technically, we use first-order logic with transitive-closure to express configurations and their properties in a parametric way. Formally, we assume that there is a set of predicate symbols $P$ for every analyzed program each with a fixed arity. For example, Table 1 contains predicates for partial semantics of Java. The predicates above the double-line separator are internal $3VMC$ predicates. The predicates below the separator are defined by the user who wishes to capture the thread-semantics of Java.

*School of Computer Science, Tel-Aviv University, Tel-Aviv 69978, Israel Tel: +972-3-6407606, Fax: +972-3-6406761 ; yahave@math.tau.ac.il

```
public class Client implements Runnable {
  public void run() {
    while (true) {
      synchronized(lock) {
        // do some critical stuff
      }
    }
  }
}
public class Main {
  public static void main() {
    Thread t1 = new Thread(new Client());
    Thread t2 = new Thread(new Client());
    Thread t3 = new Thread(new Client());
    t1.start();
    t2.start();
    t3.start();
  }
 }
```

Figure 1: Mutual exclusion example program.

- The unary predicate *isthread*$(t)$ is used to denote the objects that are threads. (e.g., in Java - instances of a subclass of `class Thread`).
- For every potential program-location (program label) *lab* of a thread $t$, there is a unary predicate $at[lab](t)$ which is true when $t$ is at *lab*. These predicates will be referred to as *location predicates*.
- For every class field and function parameter `fld`, a binary predicate $rvalue[fld](o_1, o_2)$ records the fact that the `fld` of the object $o_1$ pointing to the object $o_2$.
- The predicates *held_by*$(l, t)$, *blocked*$(t, l)$ and *waiting*$(t, l)$ model possible relationships between locks and threads.

Configurations are depicted as directed graphs. Each individual from the universe is displayed as a node. A unary predicate $p$ which holds for an individual (node) $u$ is drawn inside the node $u$. The name of a node is written inside angle brackets. Node names are only used for ease of presentation and do not affect the analysis. A true binary predicate $p(u_1, u_2)$ is drawn as directed edge from $u_1$ to $u_2$ labeled with the predicate symbol. Figure 2 shows a configuration which consists of 4 threads competing for an exclusive shared resource.

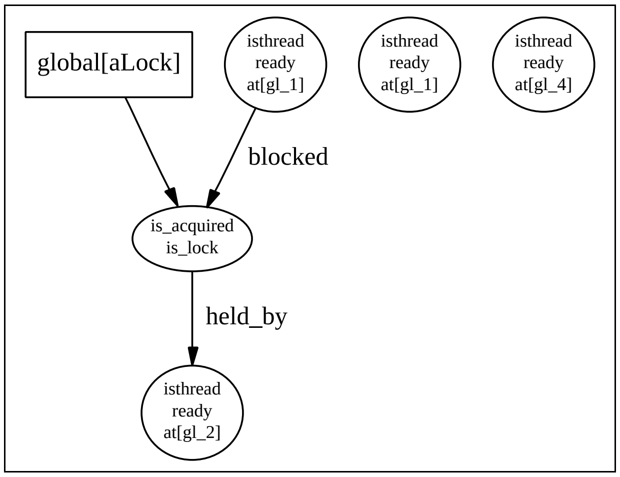

Figure 2: A concrete configuration $C_2{}^\natural$.

| **Predicates** | **Intended Meaning** |
|---|---|
| $isthread(t)$ | $t$ is a thread |
| $ready(t)$ | $t$ is ready for scheduling |
| $\{at[lab](t) : lab \in Labels\}$ | thread $t$ is at label $lab$ |
| $\{rvalue[fld](o_1, o_2) : fld \in Fields\}$ | field $fld$ of the object $o_1$ points to the object $o_2$ |
| $held\_by(l, t)$ | the lock $l$ is held by the thread $t$ |
| $blocked(t, l)$ | the thread $t$ is blocked on the lock $l$ |
| $waiting(t, l)$ | the thread $t$ is waiting on the lock $l$ |

Table 1: Predicates for partial Java semantics.

$3VMC$ conservatively represent multiple configurations using a single logical structure but with an extra truth-value 1/2 denoting values which may be 1 and may be 0. The values 0 and 1 are called *definite values* where the value 1/2 is called *indefinite value*.

## 3 Three-Valued Models

A three-valued model defines the behavior of the program to be verified. It defines the effect of *actions* and how actions are applied to configurations. A three-valued model consists of the following:

- Declarations
- Action Definitions
- Thread Types Definitions
- Properties
- Output Modifiers

Figure 3 shows the TVM file for the mutual exclusion example program.

## 3.1 Declarations

The declarations section of a three-valued model may consist of declarations of *core predicates*, *intrumentation predicates*, sets and consistency rules. The reader is referred to [3] for more details.

$3VMC$ automatically defines the predicates above the two-line separator of Table 1. In addition, $3VMC$ defines the set *Labels* which contains all labels in the program.

## 3.2 Action Definitions

Informally, an action is characterized by the following kinds of information:

- A *precondition* under which the action is *enabled* expressed as logical formula. This formula may also include a designated free variable $t_r$ to denote the "scheduled" thread on which the action is performed. Our operational semantics is non-deterministic in the sense that many actions can be enabled simultaneously and one of them is chosen for execution. In particular, it selects the scheduled thread by an assignment to $t_r$.

- An *enabled* action creates a new configuration where the interpretations of every predicate $p$ of arity $k$ is determined by evaluating a formula $\varphi_p(v_1, \ldots, v_k)$ which may use $v_1, \ldots, v_k$ and $t_r$ as well as all other predicates in $P$.

A program statement may be modeled by several alternative actions corresponding to the different behaviors of the statement. A single action to be taken is determined by evaluation of the preconditions.

Technically, an action consists of the following:

- Title - (**%t**) a textual title for the action.

- Focus formulae (**%f**) - focus formulae for the action, applied before the precondition is evaluated.

```
/***
* mutex.tvm
* mutual exclusion examples for unbound number of threads.
* based on the example from "symmetry and model checking"
**/
/*********************** Sets *****************************/
%s FieldsAndParameters { }
%s Globals { aLock }
/**************** Predicates **************/
#include "con_pred.tvm"
#include "shape_pred.tvm"
%p property_occurred()
%%
/******************** Actions ******************/
#include "con_stat.tvm"
%action verifyProperty() {
%p!property_occurred()
%message(E(t_1,t_2) (t_1 != t_2) & at[gl_2](t_1) & at[gl_2](t_2))
        -> "mutual exclusion may be violated"
}
/**************** Program ********************/
%%
/*********************************************/
/************** Threads ****************/
%thread main {
        gl_1 blockLockGlobal(aLock) gl_1               // lock(aLock)
        gl_1 succLockGlobal(aLock) gl_2
        gl_2 skip() gl_3
        gl_3 unlockGlobal(aLock) gl_4                 // unlock(aLock)
}
%%
/********************************************/
/************** Claims ****************/
verifyProperty()
```

Figure 3: TVM file for mutual exclusion example.

- Precondition formula (**%p**) - evaluated to determine if the action is *enabled* (could be applied) for a given configuration. The precondition formula may have free variables, in which case, the action is applied for every assignment that may satisfy the formula. A special free variable $t_r$ denotes

```
%thread main {
        $gl_1$ blockLockGlobal(aLock) $gl_1$              // lock(aLock)
        $gl_1$ succLockGlobal(aLock) $gl_2$
        $gl_2$ skip() $gl_3$
        $gl_3$ unlockGlobal(aLock) $gl_4$               // unlock(aLock)
}
```

Figure 4: Definition of a thread type.

the “scheduled” thread on which the action is performed.

- Universe modifiers (**%new**, **%newthread**, **%retain**) - universe modifiers allow to add/remove individuals and thread-individuals to the universe of the configuration.
- Thread state modifiers (**%start**, **%stop**)- allow to start and stop a thread. An unary formula must be supplied to identify threads to be affected by the modifier.
- Analysis Control (**%halt**) - halts the analysis.
- Update formulae - update the configuration to model the effect of the action.

## 3.3 Thread Types

$3VMC$ allows the user to define thread-types. A thread-type is defined using the keyword `%thread` followed by the thread-type name and the control-flow for the thread-type. A thread-type can be later instantiated as explained in the following section.

Figure 4 shows a definition of a thread-type named *main* which consists of 4 actions.

### 3.3.1 Thread Control-flow

Control flow in a thread-type is expressed in terms of transitions. A transition consists of a source-label, an action, and a target-label. CFG nodes are implicitly defined for each label when a transition using these labels is defined. $3VMC$ supports non-determinism when more than one enabled transition exists from a single CFG node.

$3VMC$ assumes that threads are scheduled arbitrarily, and thus thread actions may be arbitrarily interleaved with actions of other threads. A special keyword **atomic** may be used to define atomic blocks as explained in Section 3.4.

The values of the predicates $at[lab](tr)$ are automatically updated when an action is applied to express the change of $at[lab](tr)$ according to the transition. It is possible to disable automatic update of $at[lab](tr)$ predicates by using the keyword **%explicitat**in the action definition header. When explicit update is enable (automatic update disabled), the user should provide predicate-update formulae for $at[lab](tr)$ as part of the action's update formulae.

#### 3.3.2 Instantiation of Threads

A new thread can be instantiated using the universe-modifier **%newthread**. The universe modifier **%newthread** takes a name of a thread-type as parameter and create a new thread individual of the given type. The predicate $isNew(v)$ evaluates to true for the newly created thread node. The predicate $isNew(v)$ can be used by the action's update formulae. A created thread is initially not ready for scheduling.

#### 3.3.3 Thread Scheduling

The thread-state modifiers **%start** and **%stop** allow to add thread control in an action. The modifiers require an unary formula to identify threads to be affected by the modifier. The unary predicate *ready*$(v)$ evaluates to *true* for thread individuals that are ready for scheduling.

#### 3.3.4 Analysis Control

The **%halt** modifier allows an action to be used to halt the analysis. When an action with the **%halt** modifier is applied, the analysis terminates *immediately.*

### 3.4 Atomic Blocks

The keyword **atomic** is used to define a sequence of actions that should be performed atomically. An atomic block is defined using the atomic keyword followed by the block in curly braces. An atomic block should have a single exit.

### 3.5 Assertions

The keywords **%assert** and **hardassert** can be used to define assertion statements. If the formula $f$ supplied to **%assert(f)** evaluates to *true* or *unknown*, the statement has no effect. If the formula evaluates to *false*, the statement will yield an error. The **%hardassert(f)** statement is similar to **%assert(f)**, but yields an error when $f$ evaluates to *unknown* in addition to *false.*

Note that assertion statements simply save the need for declaration of *halting actions.*

```
%t = { s_t(main) }
%n = { u }
%p = {
     sm = { s_t: 1/2 }
     ready = { s_t }
     is_lock = { u }
     global[aLock] = { u }
}
```

Figure 5: TVS file for mutual exclusion example.

### 3.6 Properties

Properties are implemented more generally as *global actions*. Global actions are evaluated on every step of the analysis regardless of current active threads, current location and current thread scheduling. A global action may perform any required action, but it is common to use it only for evaluating a required property.

### 3.7 Output Modifiers

The output modifiers section allows the user to define which configurations are written to the output file. The keywords **%include** and **%exclude** are used to define configurations that should be included in the output or excluded from it. **%include** and **%exclude** take a closed formula as a parameter. Only configurations for which the formula may evaluate to *true* are included/excluded. (NYI)

## 4 Initial Configurations

A new set of nodes denoted by %t corresponds to the threads in the initial configuration. For example: %t = { $s_t(main)$ } defines a new thread instance of thread-type "main". The name of the instance is "$s_t''$. Unless explicitly stated by the $ready(v)$ predicate, the thread is assumed to be not ready for scheduling. The thread starts execution from its entry label. You may create a configuration in which a thread starts at a given label, by supplying the initial label as a second parameter - $s_t(main, gl_3)$.

Figure 5 shows an initial configuration for the mutual exclusion example. This initial configuration consists of a summary node of threads, and a single lock. Note that the thread node *s_t* is given the value 1/2 for the predicate **sm** to denote the fact that it is a summary node.

```
initialize($C_0$) {
    for each $C \in C_0$
        push(stack,C)
}
explore() {
  while stack is not empty {
    $C$ = pop(stack)
    verify(C)
    if not $member(C, stateSpace)$ {
       $stateSpace' = stateSpace \cup \{C\}$
       for each action $ac$
         for each $C'$ such that $C \Rightarrow_{ac} C'$
           push(stack,$C'$)
    }
  }
}
```

Figure 6: State space exploration.

## 5 Property Verification

### 5.1 Safety Properties

Verification of safety properties is performed by exploring the state-space, identifying configurations that may violate the property requirements.

Figure 6 shows a depth-first search algorithm for exploring a state-space. For each configuration $C$ such that $C$ is not already a *memeber* of the *state-space*, we explore every configuration $C'$ that can be produced by applying some action to the current configuration $C$. Note that the $verify$ step checks if the current configuration $C$ satisfies the property requirements.

The $verify$ step actually executes all *global actions* defined by the user. The *global actions* can be used issue messages or to halt analysis when a property is violated.

### 5.2 LTL Properties

$3VMC$ supports verification of general LTL properties by using Buchi automata to represent the (negation of the) property to be verified.

$3VMC$ takes as input a `BUC` file that models the Buchi automaton for the negation of the property to be verified. Currently, $3VMC$ does not support automatic construction of the Buchi automaton from an LTL formula. The Buchi automaton should be supplied by the user using the `BUC` file.

```
%i i_1() = E(v) isthread(v)
%i i_2() = E(v) E(l) isthread(v) & blocked(v,l)
%%
%buchi example {
  ->(b1)
  (b1) i_1 b2
  b2 i_2 b3
  b3 i_1 b4
  b4 i_2 b5
  b5 i_1 (b1)
}
```

Figure 7: BUC file defining a Buchi automata.

A `BUC` file consists of two parts - the first section of the file defines the instrumentation predicates to be used as the vocabulary of the automaton, the second section defines automaton states and transitions (and the initial state). The sections are separated by a special separator (%%).

Vocabulary instrumentation predicates are nullary predicates defined in the same manner as in the TVM file.

Automaton is described using transition as seen in Figure 7.

Accepting states of the Buchi automaton are written as $(s)$, an accepting state should always appear inside parentheses. The initial state of the automaton should be defined in the first line of automaton description using $\rightarrow$. Other transitions are defined by triples of state-label-state, where the label corresponds to one of the vocabulary predicates.

## 6 Summary

## A On-line Resources

$3VMC$ uses the *GraphViz* package to display results. GraphViz can be obtained at: http://www.research.att.com/sw/tools/graphviz/download.html

Additional examples and updated version of $3VMC$ can be downloaded from the author's home-page at: http://www.cs.tau.ac.il/ yahave

### Acknowledgment

Thanks to Nurit Dor, Tal Lev-Ami, Noam Rinetskey and Alex Warshovski.

```
%action skip() {
   %t"skip ()"
   %p!property_occurred()
}
%action succLockGlobal(glb) {
   %t"SuccesfulLocGlobal (" + glb + ")"
   %f{ is_acquired(l), global[glb](l) }
   %p!property_occurred() & global[glb](l) & (!is_acquired(l) | held_by(l, tr))
   {
   held_by(l_1, t_1) =
          (held_by(l_1, t_1) & (t_1 != tr | l_1 != l)) |
          (t_1 == tr & l_1 == l)
   blocked(t_1,l_1) = blocked(t_1,l_1) & ((t_1 != tr) | (l_1 != l))
   is_acquired(l_1) = is_acquired(l_1) | (l_1 == l)
   }
}
%action blockLockGlobal(glb) {
   %t"blockLocGlobal (" + glb + ")"
   %f{ global[glb](l) & is_acquired(l) }
   %p!property_occurred() & global[glb](l) & is_acquired(l) & !held_by(l,tr)
   {
   blocked(t_1, l_1) =
          (blocked(t_1, l_1) & (t_1 != tr | l_1 != l)) |
          (t_1 == tr & l_1 == l)
   }
}
%action unlockGlobal(glb) {
   %t"unLocGlobal (" + glb + ")"
   %f{ is_acquired(l), global[glb](l) }
   %p!property_occurred() & global[glb](l)
   {
   held_by(l_1, t_1) = held_by(l_1, t_1) & (t_1 != tr | l_1 != l)
   is_acquired(l_1) = is_acquired(l_1) & (l_1 != l)
   }
}
```

Figure 8: TVM file for simple concurrency semantics.

## References

[1] E.M. Clarke, O. Grumberg, and D. Peled, *Model checking*, MIT Press, 1999.

[2] G. J. Holzmann, *Proving properties of concurrent systems with SPIN*, Proc. of the 6th Int. Conf. on Concurrency Theory (CONCUR'95) (Berlin, GER), LNCS, vol. 962, Springer, August 1995, pp. 453–455.

[3] T. Lev-Ami and M. Sagiv, *TVLA: A framework for Kleene based static analysis*, SAS'00, Static Analysis Symposium, Springer, 2000, Available at http://www.math.tau.ac.il/∼tla.

[4] M. Sagiv, T. Reps, and R. Wilhelm, *Parametric shape analysis via 3-valued logic*, Symp. on Princ. of Prog. Lang., 1999.

[5] E. Yahav, *Verifying safety properties of concurrent Java programs using 3-valued logic*, To appear in Proc. of POPL '01, January 2001. Available at http://www.math.tau.ac.il/∼yahave/popl01.ps.