A Memory Management Mechanism for MSVL

Kai Yang, Zhenhua Duan, and Cong Tian

ICTT and ISN Lab, Xidian University, Xi’an 710071, China
kaiyang@stu.xidian.edu.cn, zhenhua.duan@126.com, ctian@mail.xidian.edu.cn

Abstract. This paper presents a memory management mechanism for programs of Modeling, Simulation and Verification Language (MSVL) which is a subset of Projection Temporal Logic (PTL) with framing technique. Framing operator is defined in MSVL and is concerned with the persistence of the values of variables from one state to another. Based on framing technique, we implement a memory management mechanism for MSVL programs. In short, memory can be allocated and released dynamically according to the framing operator. As a result, the efficiency can be improved and the memory can be used more effectively when MSVL programs are executed in MSV which is a toolkit developed for the purpose of modeling, simulation and verification of MSVL programs.

Keywords: Temporal logic · MSVL · Framing technique · Memory management

1 Introduction

Modeling, Simulation and Verification Language (MSVL) is a subset of Projection Temporal Logic (PTL) with framing technique [5,6,9]. It can be used to simulate, model and verify software and hardware systems [4,7]. A toolkit named MSV has been developed in C++ for the purpose of modeling, simulation and verification of MSVL programs. Further, MSVL programs are executed by interpreting in MSV. Symbol table is a key module in MSV, which is employed to simulate the stack and heap in a compiler, namely, saving information of variables during the execution of an MSVL program. What we care about is how to keep the size of the symbol table as small as possible to save memory.

Memory management is a technique caring about the memory of a computer when a system is running. It focuses on how to allocate and release memory resource efficiently [10,15,16]. In a program, the compiler or interpreter must be told explicitly or implicitly when a memory cell should be allocated or released. A good memory management mechanism is vital for a practical programming language.

In a conventional programming language such as C or JAVA [2,11–13], as we all know, the system will allocate a memory cell for a dynamic variable

This research is supported by the NSFC Grant No. 61133001, 61272117, 61322202, 61420106004, and 91418201.

© Springer International Publishing Switzerland 2015

DOI: 10.1007/978-3-319-17404-4_12
when it is declared and maintain the cell within the scope of the variable. If a variable has not been assigned a new value, the current value of the variable is kept. However, the situation is different in a temporal logic programming language such as MSVL. A temporal logic program is executed over a sequence of states and the values of variables do not inherit their old values automatically. Therefore, a framing technique is introduced to deal with this problem. Framing is concerned with how the value of a variable from one state can be carried to the next one.

Introduction of framing technique to temporal logic programming is motivated by both practical and theoretical aspects: improving the efficiency of a program and synchronizing communication for parallel processes [3,6,17]. A variable in a MSVL program can be framed, non-framed or framed over a subinterval. Intuitively, if a variable is framed, it always keeps its old value over an interval if no assignment to it is encountered. Otherwise, the value of the variable will be nil (not defined) if no assignment to it is encountered. Obviously, if a variable is non-framed, its value in the history will not be referenced and MSV needs not maintain a memory cell for it to save memory.

In our algorithm, in brief, MSV allocates and releases memory according to variables' scopes similar to other programming languages. To this end, the scopes of framed dynamic variables and non-framed dynamic variables are given respectively in our method. Obviously, the scope of a static variable is the whole interval, hence, static variables will be kept in the symbol table from the beginning to the end of the execution. MSV allocates memory unit for a dynamic variable (adding the variable to the symbol table) framed or non-framed at the beginning of the its scope and releases the memory unit (removing the variable from the symbol table) of it when its scope ends. Through our method, MSV will release the memory of a useless variable timely and as a consequent the memory will be saved.

The remainder of this paper is organized as follows. In Sect. 2, the framing technique in temporal logic programming is briefly introduced. The syntax and semantics of the MSVL and normal form of a MSVL program are given in Sect. 3. Section 4 presents a memory management algorithm for MSVL programs. A case study is given in Sect. 5 and conclusion is drawn in Sect. 6.

2 Framing Techniques

Framing is concerned with whether or not the value of a variable should be persisted over an interval. Intuitively, if a variable $x$ is framed in an interval, the value of $x$ in the previous will be inherited over an interval if there are no new assignments to $x$ are encountered. There are state framing ($lbf$) and interval framing (frame) operators. Specifically, when a variable is framed at a state, its value remains unchanged if no assignment is encountered at that state. A variable is framed over an interval if it is framed at every state over the interval. The following are the definitions of $lbf$ and frame.
A Memory Management Mechanism for MSVL

\[ lbf(x) \overset{\text{def}}{=} \neg p_x \rightarrow \exists b : (\bigcirc x = b \land x = b) \]

\[ \text{frame}(x) \overset{\text{def}}{=} \Box (\text{more} \rightarrow \bigcirc lbf(x)) \]

\[ \text{frame}(x_1, \ldots, x_n) \overset{\text{def}}{=} \text{frame}(x_1) \land \ldots \land \text{frame}(x_n) \]

where \( b \) is a static variable and \( p_x \) an atomic proposition associated with state (dynamic) variable \( x \). \( p_x \) holds if \( x \) is assigned a new value at the current state, otherwise \( \neg p_x \) holds. In addition, \( p_x \) cannot be used for other purpose. Here, \( \neg \), \( \land \) and \( \rightarrow \) are defined as usual. \( \bigcirc x \) denotes the value of \( x \) at the previous state. For an MSVL program \( p \), \( \Box p \) means that \( p \) holds at every state over the whole interval and \( \bigcirc p \) indicates that \( p \) holds at the next state. In addition, \( \text{more} \) means that the current state is not the final state of the interval.

3 Modeling, Simulation and Verification Language

The arithmetic expression \( e \) and boolean expression \( b \) of MSVL are inductively defined as follows:

\[ e ::= n \mid x \mid \bigcirc x \mid \bigcirc x \mid e_0 \ op \ e_1 \ (\ op ::= + \mid - \mid * \mid \backslash \mid \mod) \]

\[ b ::= \text{true} \mid \text{false} \mid e_0 = e_1 \mid e_0 < e_1 \mid \neg b \mid b_0 \land b_1 \]

where \( n \) is an integer and \( x \) a variable. Some elementary statements in MSVL that are used in this paper are presented as follows. Please refer to [7] for the definition of all statements in MSVL.

\[ \begin{align*}
\text{Termination} : \text{empty} & \quad \text{Assignment} : x = e \\
P - I - \text{Assignment} : x \leftarrow e & \quad \text{Conjunction} : p \land q \\
\text{Selection} : p \lor q & \quad \text{Next} : \bigcirc p \\
\text{Always} : \Box p & \quad \text{Sequence} : p; q \\
\text{Conditional} : \text{if } b \text{ then } p \text{ else } q & \overset{\text{def}}{=} (b \rightarrow p) \land (\neg b \rightarrow q) \\
\text{While} : \text{while } b \text{ do } p & \overset{\text{def}}{=} (p \land b) \ast \land \Box (\text{empty} \rightarrow \neg b)
\end{align*} \]

where \( x \) denotes a variable, \( e \) an arbitrary arithmetic expression, \( b \) a boolean expression, and \( p \) and \( q \) programs of MSVL.

The termination statement \text{empty} means that the current state is the final state of the interval over which the program is executed. The assignment \( x = e \) denotes that the value of variable \( x \) is equal to the value of expression \( e \). Positive immediate assignment \( x \leftarrow e \) indicates that the value of \( x \) is equal to the value of \( e \) and the assignment flag \( p_x \) for variable \( x \) is \text{true}. The conjunction statement \( p \land q \) denotes that \( p \) and \( q \) are executed in a concurrent manner and share all the variables during the execution, and \( p \) and \( q \) have the same interval lengths. \( p \lor q \) means \( p \) or \( q \) is selected randomly to execute. The next statement \( \bigcirc p \) means that \( p \) holds at the next state. Intuitively, the sequence statement \( p; q \) means that program \( p \) is executed from the current state until its termination, then program \( q \) is executed. The conditional statement \text{if } b \text{ then } p \text{ else } q, \) as in the conventional programming language, means that if the condition \( b \) is evaluated \text{true} then
process \( p \) is executed, otherwise process \( q \) is executed. Statement \( \text{while } b \text{ do } p \) is the loop statement which denotes that process \( p \) will be repeatedly executed until condition \( b \) becomes \( \text{false} \).

3.1 Normal Form of MSVL Programs

Definition 1 (Normal Form of MSVL Programs). An MSVL program \( q \) is in its Normal Form if \( q \) has been rewritten in the following form:

\[
q \equiv \bigvee_{i=1}^{k} q_{ei} \land \text{empty} \lor \bigvee_{j=1}^{h} q_{cj} \land \bigcirc q_{fj}
\]

where \( k, h \geq 0 \) (\( k + h \geq 1 \)), and

- each \( q_{ei} \) and \( q_{cj} \) is either \( \text{true} \) or a state formula of the form \((x_1 = e_1) \land \ldots \land (x_l = e_l) \land \hat{p}_x \land \ldots \land \hat{p}_x \) where \( e_i \in D \) (\( 1 \leq i \leq l \)), and \( \hat{p}_x \) denotes \( p_x \) or \( \neg p_x \), \( l \geq 0 \), \( m \geq 0 \), and \( l + m \geq 1 \).

Specially, we call \( q_{cj} \) the present component, \( q_{fj} \) the future component, and \( q_{ei} \) the termination component in a normal form.

The following theorem has been proved in [7].

Theorem 1. Any MSVL program \( p \) can be rewritten into its normal form.

Theorem 1 tells us that for any MSVL program \( p \) there is a program \( q \) in the normal form such that \( p \equiv q \).

4 Memory Management for MSVL

As known to all, every variable has its scope in a conventional program. For instance, in a C program, for an dynamic variable declared at the beginning of a block, the scope is the block in which the name is declared. The compiler or interpreter allocates and releases memory according to the scope of a variable. When a dynamic variable is declared, the compiler will allocate a memory unit for it in the variable stack for it and maintain the memory unit until the end of the variable’s scope. Throughout it, the value of the variable may be modified by a process. In other words, the current value of the variable remains until a new assignment to it is encountered.

In order to implement memory management for MSVL programs, the scopes of variables (all variables are dynamic variables if not specified specially) in MSVL programs also need to be specified. Because the value of a framed variable will be kept in the framed interval while the value of a non-framed variable will not be taken to the next state. Informally, the scope of a framed variable is the framed interval associated with it and the scope of a non-framed variable is the state in which it is assigned.
Although there are differences among scopes, memory management for framed and non-framed variables can be dealt with in the same way with our method. Formally, the state memory management algorithm and interval memory management algorithm for MSVL programs are given as StateMM and IntervalMM. As their names show, StateMM deals with the memory allocation and release at a state while IntervalMM cares about the memory allocation and release over a whole interval. Further, considering efficiency, a symbol table in MSV is implemented with the data structure map in STL of C++ [14].

**Algorithm 1. StateMM** *(q, ST) *

**Input:**
- *q* is a disjunct in \( NF(P) \) where *P* is the program to be executed at the current state *ST* is the symbol table

**Output:**
- *ST*

```plaintext
1: if !ST.empty() then
2:     for each variable *x* in ST do
3:         if *p_x* is not in the present component of *q* then
4:             if ¬*p_x* is not in the present component of *q* then
5:                 ST.erase(*x*);
6:             end if
7:         end if
8:     end for
9: end if
10: for each state formula *x* = *e* in the present component of *q* do
11:     if !ST.find(*x*) then
12:         ST.insert(*x*);
13:         Assign(*x*, *e*);
14:     else
15:         Assign(*x*, *e*);
16:     end if
17: end for
```

In Algorithm StateMM, \( NF \) is an algorithm defined in [8] for translating a preprocessed MSVL to its normal form. \( Assign(x, e) \) assigns value *e* to variable *x*.

To execute an MSVL program, at the beginning of each state, MSV translates the program to its normal form and then one of the disjuncts in the normal form will be selected randomly to execute. Before the present component of the selected disjunct is executed, MSV checks the variables whose life cycles have ended at this state and the memory space of these variables will be released. Specifically, for each variable *x* in the symbol table (if the symbol table is not empty), MSV checks if there is *p_x* or ¬*p_x* in the present component firstly. If no, MSV removes variable *x* from the symbol table. Then the assignment statements in present component will be executed. For instance, when *x* = 3 is to be executed, MSV will add variable *x* to the symbol table and then assign 3 to *x* if the variable *x* is not in the symbol table.
Algorithm 2. IntervalMM($P$, ST)

<table>
<thead>
<tr>
<th>Input:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$ is the program to be executed</td>
</tr>
<tr>
<td>ST is the symbol table</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output:</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST</td>
</tr>
</tbody>
</table>

1: ST=$\emptyset$;
2: if existing a next state in $P$ and the program to be executed at next state is $q$ then
3: NF(q);
4: StateMM(q, ST);
5: else
6: ST.clear();
7: end if

It is known from the definition of $frame(x)$ that for a framed variable $x$, starting from the second state of the framed interval, there must be $p_x$ in the present component if $x$ is assigned a new value at current state, or $\neg p_x$ in the present component otherwise. As a consequence, in our method, the memory cell of variable $x$ will be maintained until the interval of $frame(x)$ is terminated. It is trivially correct for non-framed variables. For a non-framed variable $x$, if $x$ is not assigned at the current state, $p_x$ and $\neg p_x$ will not appear in the present component and the memory space of variable $x$ will be released in this case.

The following is a MSVL program called GCD which computes the greatest common divisor of $x$ and $y$ with the result saved in variable $g$. The program is an implementation of Euclid's well known algorithm [1].

```plaintext
frame(x, y) and (
    x <= 6 and y <= 4 and empty;
    while(x! = y) {
        if(x > y) then { x := x - y
            else { y := y - x
        }
    };
    g := x
)
```

In the above program, $x$ and $y$ are framed dynamic variables while $g$ is a non-framed dynamic variable. Variable $g$ is not assigned until the last state (state 3). Hence, its value is unspecified at state 0, 1 and 2. The computation which gives the values of the variables of each state in the program is depicted in Fig. 1. The mark (?) means that the value of the variable is unknown at the state.

According to algorithm IntervalMM, the symbol table is empty initially. Firstly, we translate the program to its normal form and obtain the present and future component of each state. Only the present components are given here because we do not care about the future components in our algorithm. $p_0^0$ stands for the present component of state 0 and so on, $p_3^3$ denotes the termination component of state 3. The left hand side of Fig. 2 shows the symbol table
before the present component of state \( i \) is executed and the right hand side of Fig. 2 shows the symbol table after the present component of state \( i \) is executed. The types of the variables are ignored here for clarity (all variables are integers actually).

\[
\begin{align*}
p_0^c & \equiv x = 6 \land p_x \land y = 4 \land p_y \\
p_1^c & \equiv x = 2 \land p_x \land y = 4 \land \neg p_y \\
p_2^c & \equiv x = 2 \land \neg p_x \land y = 2 \land p_y \\
p_3^c & \equiv x = 2 \land \neg p_x \land y = 2 \land \neg p_y \land g = 2 \land p_g
\end{align*}
\]

Furthermore, frame statements that are sequential and nested in a MSVL program can be dealt with in our method. The latter is more complex and we use the following example called LCM to illustrate this case. Firstly, the program computes the greatest common divisor of \( x \) and \( y \) with the result eventually saved in \( g \). Then \( g \) is used to compute the least common multiple of \( x \) and \( y \). The eventual result is saved in variable \( l \).

\[
frame(x, y) \text{ and } (x <= 6 \text{ and } y <= 4 \text{ and empty;})
\]

\[
frame(x1, y1) \text{ and } (x1 := x \text{ and } y1 := y;)
\]

\[
while(x != y) \{ 
    if(x > y) \text{ then } \{x := x - y\}
    else \{y := y - x\};
\]

\[
frame(g) \text{ and } (g := x;)
\]

\[
x := (x1 \times y1)/g;)
\]

\[
l := x
\]

The computation and symbol table of each state are given in Figs. 3 and 4 respectively. In the program, variables \( x \) and \( y \) are framed at the overall interval
while variables \( x_1 \) and \( y_1 \) are framed at a subinterval of the interval \( frame(x, y) \) is executed, and variable \( g \) is framed at a subinterval of the interval \( frame(x_1, y_1) \) is executed. Initially, the symbol table is empty at state 0. Variables \( x \) and \( y \) are added to the symbol table at state 0. Similarly, variables \( x_1 \) and \( y_1 \) are added to the symbol table at state 1, and variable \( g \) is added to the symbol table at state 4. Considering their scopes, variables \( x \) and \( y \) are kept in the symbol table over the whole interval. Variables \( x_1, y_1, \) and \( g \) are removed from the symbol table at state 6 because the intervals associated with \( frame(x_1, y_1) \) and \( frame(g) \) are terminated at this state.

\[
\begin{array}{cccccc}
& s0 & s1 & s2 & s3 & s4 & s5 & s6 \\
\hline
x=6 & 6 & 2 & 2 & 2 & 12 & 12 \\
y=4 & 4 & 4 & 2 & 2 & 2 & 2 \\
x_1=? & 6 & 6 & 6 & 6 & 6 & ? \\
y_1=? & 4 & 4 & 4 & 4 & 4 & ? \\
\end{array}
\]

**Fig. 3.** Computation of LCM

**Fig. 4.** Symbol tables of LCM

5 A Case Study

Matrix operation is important in digital image processing. Using a matrix to store a large image always takes up a lot of memory space. We implement an algorithm for calculating the product matrix \( C \) of two matrices \( A \) and \( B \) and the transpose matrix \( D \) of the product matrix \( C \) in MSVL. Part of the execution result in MSV is shown in Fig. 5. In the program, four two-dimensional arrays \( A, B, C \) and \( D \) are used to denote the matrices \( A, B, C \) and \( D \) respectively. Variable \( C \) is framed over the whole interval while \( A \) and \( B \) are framed in a
A Memory Management Mechanism for MSVL

Fig. 5. Matrix operation

subintervals of the interval where \textit{frame}(C) is executed, \textit{D} is framed at another subinterval of the interval where \textit{frame}(C) is executed. In the algorithm, after getting the product matrix of \textit{A} and \textit{B}, matrices \textit{A} and \textit{B} will be useless and we do not need to maintain them in the memory of for saving memory. Furthermore, the larger the size of \textit{A} or \textit{B} the more the memory will be saved. This is of great importance for processing large matrix in practice. Table 1 shows the memory that the symbol tables takes (in the old version of MSV and the new version of MSV) and the memory saved for different sizes of matrices.

Table 1. Comparison of memory usage

<table>
<thead>
<tr>
<th>Size</th>
<th>Old (Mb)</th>
<th>New (Mb)</th>
<th>Memory saved (Mb)</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 300*200</td>
<td>1.52</td>
<td>1.04</td>
<td>0.48</td>
</tr>
<tr>
<td>B 200*400</td>
<td>6.08</td>
<td>4.16</td>
<td>1.92</td>
</tr>
<tr>
<td>A 600*400</td>
<td>13.68</td>
<td>9.36</td>
<td>4.32</td>
</tr>
<tr>
<td>B 400*800</td>
<td>24.32</td>
<td>16.64</td>
<td>7.68</td>
</tr>
</tbody>
</table>

6 Conclusion

This paper presents a memory management mechanism for MSVL programs based on the framing technique. With our method, the memory can be saved and the efficiency can be improved to execute an MSVL program in MSV. In the future, the memory management of dynamic variables in functions will be considered. Besides, MSV toolkit will be further improved.
References