Extending MSVL with Semaphore

Xinfeng Shu[1] and Zhenhua Duan[2(✉)]

[1] School of Computer Science and Technology,
Xi'an University of Posts and Communications, Xi'an 710061, China
shuxf@xupt.edu.cn
[2] Institute of Computing Theory and Technology, Xidian University,
Xi'an 710071, China
zhhduan@mail.xidian.edu.cn

Abstract. Modeling, Simulation and Verification Language (MSVL) is a useful formalism for specification and verification of concurrent systems. To make it more practical and easy to use, we extend MSVL with the technique of semaphore. To do so, the mechanism of MSVL function calls is deeply analyzed. Further, the semaphore type is defined. Moreover, operations over semaphore are formalized. Finally, an example is given to illustrate how to use semaphore to solve the mutual exclusion problem.

Keywords: Temporal logic programming · Projection · Semaphore · Mutual exclusion · Concurrency

1 Introduction

Modeling, Simulation and Verification Language (MSVL) [1], an executable subset of Projection Temporal Logic (PTL) [2,3] with framing technique, is a useful formalism for specification and verification of concurrent and distributed systems [4–6]. It provides a rich set of data types (e.g., char, integer, pointer, string), data structures (e.g., array, list), as well as boolean and arithmetic expressions. Besides, MSVL supports not only the commonly used statements such as assignment, sequential, branch and loop, but also parallel and concurrent statements such as conjunct (S_1 and S_2), parallel ($S_1||S_2$) and projection ($(S_1, \ldots, S_m)$ prj S). Further, Propositional Projection Temporal Logic (PPTL), the propositional subset of PTL, has the expressiveness power of the full regular expressions [7], which enables us to model, simulate and verify the concurrent and reactive systems within a same logical system [8].

In verification of concurrent and distributed systems, an essential problem that must be dealt with is the synchronization and communication between concurrent processes. To solve the problem, some formalisms involve synchronous

This research is supported by Natural Science Foundation of Education Bureau of Shaanxi Province (No. 11JK1037), NSFC Grant Nos. 61133001, 61420106004, 91418201 and 61322202.

T.N. Dinh and M.T. Thai (Eds.): COCOON 2016, LNCS 9797, pp. 599–610, 2016.
DOI: 10.1007/978-3-319-42634-1_48

message passing (e.g., CCS [9] and CSP [10]), and some involve asynchronous channels (e.g., PROMELA [11]). As for MSVL, the communication between parallel components is based on shared variables. Furthermore, MSVL provides a synchronization construct, $await(c)$, to synchronize communication between parallel processes. The meaning of $await(c)$ is simple: it changes no variables, but keeps on waiting until the condition c becomes $true$, at which point it terminates. With this statement, the synchronization between two parallel processes can be easily achieved since another process can cause c to become $true$.

However, the mutual exclusively accessing critical resource for many concurrent processes has not solved in MSVL so far. Therefore, we are motivated to introduce the technique of semaphore [12] to MSVL. To this end, the mechanism of the function of MSVL is deeply analyzed, based on which the semaphore type is defined and the functions to initialize a semaphore variable, allocate as well as release a critical resource are also formalized. Besides, an example is given to illustrate how to use the semaphore of MSVL to solve the synchronization and mutual exclusion problem between currently processes.

The rest of paper is organized as follows. In the next section, PTL and MSVL are briefly introduced. In Sect. 3, the mechanism of MSVL functions calls is analyzed. In Sect. 4, the semaphore is introduced to MSVL. In Sect. 4.1, an example is given to illustrate how to use the semaphore of MSVL to solve the synchronization and mutual exclusion problem between currently processes.

The rest of paper is organized as follows. In the next section, PTL and MSVL are briefly introduced. In Sect. 3, the mechanism of MSVL functions calls is analyzed. In Sect. 4, the semaphore is introduced to MSVL. In Sect. 4.1, an example is given to illustrate how to program with semaphore. Finally, conclusions are drawn in Sect. 5.

2 Preliminaries

2.1 Projection Temporal Logic

In this subsection, the syntax and semantics of Projection Temporal Logic (PTL) are briefly introduced. More details can be found in paper [2].

Syntax. Let $Prop$ be a countable set of atomic propositions and V a countable set of typed variables. $B = \{true, false\}$ represents the boolean domain. D denotes the data domain of the underlying logic. The terms e and formulas P of PTL are inductively defined as follows:

$$e ::= d \mid a \mid x \mid \bigcirc e \mid f(e_1, \ldots, e_m)$$
$$P ::= p \mid e_1 = e_2 \mid \rho(e_1, ..., e_m) \mid \neg P \mid P_1 \wedge P_2 \mid \exists\, vP \mid \bigcirc P \mid (P_1, \ldots, P_m)\ prj\ P$$

where $d \in D$ is a constant, $a \in V$ is a static variable, $x \in V$ is a dynamic variable, $v \in V$ is either a static variable or a dynamic one; $p \in Prop$ is an atomic proposition; f is a function and ρ is a predicate both defined over D.

Abbreviation. The conventional constructs $true$, $false$, $\wedge$, $\rightarrow$ as well as $\leftrightarrow$ are defined as usual. Furthermore, we use the following abbreviations:

$$\begin{array}{rcl@{\qquad}rcl}
\varepsilon & \stackrel{\text{def}}{=} & \neg\bigcirc true & \overline{\varepsilon} & \stackrel{\text{def}}{=} & \neg\,\varepsilon \\
\odot P & \stackrel{\text{def}}{=} & \neg\bigcirc\neg P & P;Q & \stackrel{\text{def}}{=} & (P, Q)\ prj\ \varepsilon \\
\Diamond P & \stackrel{\text{def}}{=} & true; P & len(n) & \stackrel{\text{def}}{=} & \bigcirc^n \varepsilon \\
\Box P & \stackrel{\text{def}}{=} & \neg\Diamond\neg P & keep(P) & \stackrel{\text{def}}{=} & \Box(\,\overline{\varepsilon} \rightarrow P) \\
skip & \stackrel{\text{def}}{=} & \bigcirc\,\varepsilon & halt(P) & \stackrel{\text{def}}{=} & \Box(\,\varepsilon \leftrightarrow P)
\end{array}$$

$$\forall v P \stackrel{\text{def}}{=} \neg\exists v \neg P \qquad fin(P) \stackrel{\text{def}}{=} \Box(\varepsilon \rightarrow P)$$
$$P||Q \stackrel{\text{def}}{=} ((P;true) \wedge Q) \vee (P \wedge (Q;true)) \vee (P \wedge Q)$$

Semantics. A state s is a pair of assignments (I_p, I_v), which I_p assigns each atomic proposition $p \in Prop$ a truth value in B, whereas I_v assigns each variable $v \in V$ a value in D. An interval (i.e., model) σ is a non-empty sequence of states. The length of σ, denoted by $|\sigma|$, is ω if σ is infinite, or the number of states minus one if σ is finite. We use notation $\sigma_{(i..j)}$ to mean that a subinterval $<s_i, \ldots, s_j>$ of σ with $0 \leq i \preceq j \leq |\sigma|$. The *concatenation* of a finite interval $\sigma = <s_0, \ldots, s_{|\sigma|}>$ with another interval $\sigma' = <s'_0, \ldots, s'_{|\sigma'|}>$ (may be infinite) is denoted by $\sigma \bullet \sigma'$ and $\sigma \bullet \sigma' = <s_0, \ldots, s_{|\sigma|}, s'_0, \ldots, s'_{|\sigma'|}>$. Let $\sigma = <s_0, s_1, \ldots, s_{|\sigma|}>$ be an interval and $r_1, \ldots, r_h$ be integers ($h \geq 1$) such that $0 \leq r_1 \leq r_2 \leq \ldots \leq r_h \preceq |\sigma|$. The projection of σ onto $r_1, \ldots, r_h$ is the interval (called projected interval) $\sigma \downarrow (r_1, \ldots, r_h) = <s_{t_1}, \ldots, s_{t_l}>$, $(t_1 < t_2 < \ldots < t_l)$, where $t_1, \ldots, t_l$ is obtained from $r_1, \ldots, r_h$ by deleting all duplicates. For example, $<s_0, s_1, s_2, s_3, s_4, s_5> \downarrow (0, 2, 2, 2, 4, 4, 5) = <s_0, s_2, s_4, s_5>$.

An interpretation, as for PTL, is a triple $\mathcal{I} = (\sigma, i, j)$, where σ is an interval, $i \in N_0$ and $j \in N_\omega$, and $0 \leq i \preceq j \leq |\sigma|$. We use notation (σ, i, j) to mean that a term or a formula is interpreted over a subinterval $<s_i, \ldots, s_j>$ of σ with the current state being s_i. Then, for every term e, the evaluation of e relative to $\mathcal{I}$, denoted by $\mathcal{I}[e]$, is defined by induction on the structure of the term as follows:

$$\begin{aligned}
&\mathcal{I}[d] && = d, \text{ if } d \in D \text{ is a constant value}\\
&\mathcal{I}[a] && = I_v^i[a] = I_v^0[a], \text{ if } a \text{ is typed static variable}\\
&\mathcal{I}[x] && = I_v^i[x], \text{ if } x \text{ is typed dynamic variable}\\
&\mathcal{I}[\bigcirc e] && = \begin{cases} (\sigma, i+1, j)[e], & \text{if } \ i < j \\ nil, & \text{otherwise} \end{cases}\\
&\mathcal{I}[f(e_1, \ldots, e_m)] && = \begin{cases} nil, \text{ if } \mathcal{I}[e_h] = nil \text{ for some } h(1 \leq h \leq m) \\ f(\mathcal{I}[e_1], \ldots, \mathcal{I}[e_m]), \text{ otherwise} \end{cases}
\end{aligned}$$

The satisfaction relation ($\models$) for PTL formulas is inductively defined as follows:

$\mathcal{I} \models p$ iff $I_p^i[p] = true$, for any given atomic proposition p
$\mathcal{I} \models \rho(e_1, ..., e_m)$ iff ρ is a primitive predicate other than $=$ and, for all $h(1 \leq h \leq m)$, $\mathcal{I}[e_h] \neq nil$ and $\rho(\mathcal{I}[e_1], \ldots, \mathcal{I}[e_m]) = true$
$\mathcal{I} \models e_1 = e_2$ iff $\mathcal{I}[e_1] = \mathcal{I}[e_2]$
$\mathcal{I} \models \neg P$ iff $\mathcal{I} \not\models P$
$\mathcal{I} \models P \wedge Q$ iff $\mathcal{I} \models P$ and $\mathcal{I} \models Q$
$\mathcal{I} \models \exists v P$ iff $(\sigma', i, j) \models P$ for some interval σ', $\sigma_{(i..j)} \stackrel{v}{=} \sigma'_{(i..j)}$
$\mathcal{I} \models \bigcirc P$ iff $i < j$ and $(\sigma, i+1, j) \models P$
$\mathcal{I} \models (P_1, ..., P_m)\ prj\ Q$ iff there exist integers $i = r_0 \leq \ldots \leq r_{m-1} \leq r_m \preceq j$ such that $(\sigma, r_{l-1}, r_l) \models P_l$ for all $1 \leq l \leq m$, and $(\sigma', 0, |\sigma'|) \models Q$ for one of the following σ':
(1) $r_m < j$ and $\sigma' = \sigma \downarrow (r_0, \ldots, r_m) \bullet \sigma_{(r_m+1..j)}$
(2) $r_m = j$ and $\sigma' = \sigma \downarrow (r_0, \ldots, r_h)$ for some $0 \leq h \leq m$

2.2 Modeling, Simulation and Verification Language

Modeling, Simulation and Verification Language (MSVL) is an executable subset of PTL. In the following, we briefly introduce the kernel of MSVL. For more deals, please refer to literature [1].

Expression. The arithmetic expressions e and boolean expressions b of MSVL are inductively defined as follows:

$$e ::= d \mid x \mid \bigcirc e \mid \ominus e \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1/e_2 \mid e_1 \,\%e_2$$
$$b ::= true \mid false \mid \neg b \mid b_1 \wedge b_2 \mid e_1 = e_2 \mid e_1 \leq e_2$$

where d is is an integer or a floating point number; $x \in V$ is a static or dynamic variable; $\bigcirc e$ ($\ominus e$) refers to the value of expression e at the next (previous) state.

Statement. The elementary statements in MSVL are defined as follows:

(1) Immediate Assign $x \Leftarrow e \stackrel{\text{def}}{=} x = e \wedge p_x$
(2) Unit Assignment $x := e \stackrel{\text{def}}{=} \bigcirc x = e \wedge \bigcirc p_x \wedge skip$
(3) Conjunction $S_1 \; and \; S_2 \stackrel{\text{def}}{=} S_1 \wedge S_2$
(4) Selection $S_1 \; or \; S_2 \stackrel{\text{def}}{=} S_1 \wedge S_2$
(5) Next $next \; S \stackrel{\text{def}}{=} \bigcirc S$
(6) Always $always \; S \stackrel{\text{def}}{=} \Box S$
(7) Termination $empty \stackrel{\text{def}}{=} \neg\bigcirc true$
(8) Skip $skip \stackrel{\text{def}}{=} \bigcirc \varepsilon$
(9) Sequential $S_1; S_2 \stackrel{\text{def}}{=} (S_1, S_2) \; prj \; \varepsilon$
(10) Local $exist \; x : S \stackrel{\text{def}}{=} \exists x : S$
(11) State Frame $lbf(x) \stackrel{\text{def}}{=} \neg \mathrm{af(x)} \rightarrow \exists \mathrm{b}{:}(\ominus \mathrm{x} = \mathrm{b} \wedge \mathrm{x} = \mathrm{b})$
(12) Interval Frame $frame(x) \stackrel{\text{def}}{=} \Box(\bar{\varepsilon} \rightarrow \bigcirc(lbf(x)))$
(13) Projection $(S_1, \ldots, S_m) \; prj \; S$
(14) Condition if b then S_1 else $S_2 \stackrel{\text{def}}{=} (b \rightarrow S_1) \wedge (\neg b \rightarrow S_2)$
(15) While while b do $S \stackrel{\text{def}}{=} (b \wedge S)^{\star} \wedge \Box(\varepsilon \rightarrow \neg b)$
(16) Await $await(b) \stackrel{\text{def}}{=} \bigwedge_{x \in V_b} frame(x) \wedge \Box(\varepsilon \leftrightarrow b)$
(17) Parallel $S_1 || S_2 \stackrel{\text{def}}{=} ((S_1; true) \wedge S_2) \vee (S_1 \wedge (S_2; true)) \vee (S_1 \wedge S_2)$

where the immediate assignment $x \Leftarrow e$, unit assignment $x \Leftarrow e$, $empty$, $lbf(x)$ and $frame(x)$ are basic statements, and the left are composite ones.

3 Mechanism of MSVL Function

For convenience of modeling for complex software and hardware systems, MSVL takes the divide-and-conquer strategy and employees functions as the basic components like C programming language does. The general grammar of MSVL function is as follows:

function funcName(in_type_1 x_1, ..., in_type_m x_m,
out_type_1 y_1,..., out_type_n y_m, *return_type* RValue)
{ S } //Function body

where *function* is the keyword to declare a function; funcName is the identifier by which the function can be called; in_type_i x_i (out_type_i y_i) (as many as needed) specifies the i-th input (output) parameter consisting of a type followed by a variable identifier; *return_type* specifies the return type of the function; S, usually a compound MSVL statement, defines the operations inside the function. A function with no input (output) parameters or return value is allowed.

Parameter passing in MSVL is similar to that in C, i.e. all function arguments are passed by values (call-by-value). With call-by-value, the actual argument expression is evaluated, and the resulting value is bound to the corresponding formal parameter in the function. Even if the function may assign a new value to its formal parameter, only its local copy is assigned and anything passed into a function call is unchanged in the callers scope when the function returns. Furthermore, the pointer type is also supported by MSVL, which allows both caller and callee to access and modify a same variable.

To make MSVL more practical and useful, MSVL provides two kinds of function calls, namely internal call and external call [13]. The grammar of the internal call is the *default one*, i.e. $funcName(v_1, \ldots, v_n)$, and the grammar of external call is the general function call statement with the prefixed constraint *ext*, i.e. *ext* $funcName(v_1, \ldots, v_n)$.

For instance, Example 1 is an MSVL program to compute $(1+2+3)*2$. The program consists of two functions *main* and $GetSum1$, which function *main* is the entry of the program and function $GetSum1$ is to compute the sum of $1 + .. + n$. Within function *main*, the function call statement marked with (1) is an internal call, whereas the one marked with (2) is an external call.

Example 1. Program to compute $(1 + 2 + 3) * 2$

```
function GetSum(int n, int *rst) {
    frame(i) and (
        int i and i<== 1 and empty;
        *rst<== 0 and empty;
        while(i<= 3){ *x:= *x+i and i:=i+1 }
    )
};
function main(){
    frame(sum) and (
        int sum and sum<== 0 and skip;
        GetSum(3, &sum);        //(1) Internal function call
        ext GetSum(3, &sum);    //(2) External function call
        sum:=sum*2
    )
};
main()
```

In the following, we make a deep analysis on the di erence between the two function calls with Example 1.

Internal Function Call. Internal call means the execution of the called function is transparent to the calling function. The whole model of the calling function is the concatenation of the sub-model of the statements before the function call, the sub-model of the called function and the sub-model of statements after the function call.

For instance, if we remove the statement (2) from the program in Example 1, the execution model of the left program is given in Fig. 1, where the states of the model is marked as the labels on the edges between the nodes. It is the concatenation of the sub-models of statement •int sum and sum==0 and skipŽ, function GetSum1, and statement •sum:=sum*2Ž.

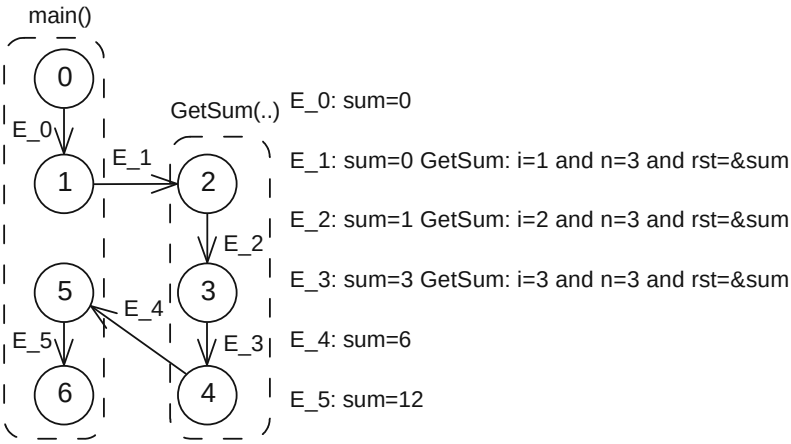

Fig. 1. The execution model of internal function call

Internal function call has a better support for modeling concurrent systems. The basic idea is to describe each concurrent component by an MSVL function and execute them in parallel. The interaction among concurrent components can be realized by shared variables among functions. However, internal call also has its negative side. The parallel execution of functions may lead to the functions a ecting each other and getting wrong result. So, we must be very careful to estimate and avoid the error interactions among parallel functions in system modeling.

External Function Call. External call means the execution of the called function is unseen to the calling function. From the view point of the function caller, it focuses on obtaining the computing result and ignores the model over which the called function is executed. The execution of an external called function is completely isolated from the function caller.

For instance, if we remove the statement (1) from the program in Example 1, the execution model of the left program is given in Fig. 2. Although function *GetSum*() takes 4 states to compute $(1 + 2 + 3)$, the model of *GetSum*() is abandoned in case of the computing result is obtained. Thus, the execution of external call nearly has no affection on the model of the function caller except for assigning the result 6 to the variable *sum*.

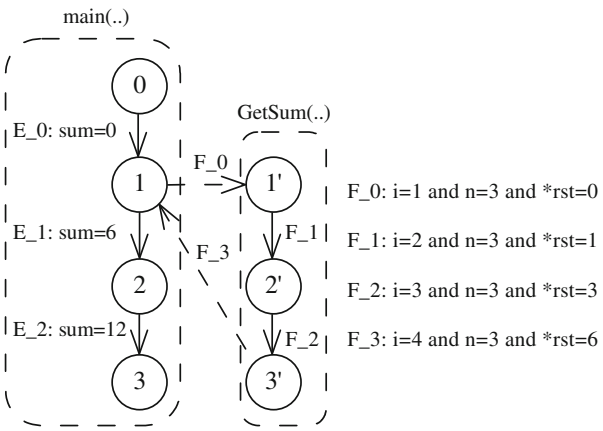

Fig. 2. The execution model of external function call

The external function call completely encapsulates the inner variables (data) and the program logic of the called function, which provides a new system modeling way at a higher level of abstraction. Compared with the internal function call, the external call is used to model the sub-system which has no interaction with function caller during its execution. In most software/hardware system, such sub-systems is the major part. The external function call greatly decreases the complexity of system modeling and helps to ensure the correctness of the system model.

4 Introduction of Semaphore to MSVL

The semaphore [12] is a key technique widely used in concurrent system development, e.g. Operating System and Web Application, to provide mutual exclusion accessing critical resource for many concurrent processes. Intuitively, a semaphore is the entity representing a kind of critical resources. Further, two atomic operations (i.e., a sequence of instructions providing a specific function and its running cannot be interrupted.) *wait* and *signal* are defined over semaphore to allocate and release a unit of resource respectively. In compute system, the realization of atomic operation must be supported by hardware. As for MSVL, we have no such support and hence can only search for an alternative

approach to realize semaphore. In the following, we firstly give the definition of semaphore, and then formalize the operations over it.

Semaphore. Let $n \in N_0$ be the number of processes using the semaphore. The *semaphore* is a struct type defined in MSVL as follows:

```
semaphore(n) def= struct {
                      int locked and
                      int value and
                      int procNum and
                      int curApp[n] and
                      int runAuth[n] and
                      list(int) procQue
                  }
```

where *locked* is the status of the lock for accessing the critical section of the semaphore (0 denotes free, 1 denotes locked); *value* denotes the number of resources; *procNum* saves the max number of processes applying for the resource; arrays *curApp* and *runAuth* record the processes currently applying for and authorized to use the resource respectively; queue *procQue* keeps the blocked processes in arriving sequence.

To solve the problem that many concurrent processes may apply for the resource at the same time, we assign each process a unique *id* ($0 \leq id < n$) and employ array members *curApp*[*id*] and *runAuth*[*id*] to handle the application and authorisation of process *id* respectively.

Semaphore is a parameterized type. To define a semaphore variable, the number of processes to use the resource must be estimated previously. For example, if the process number is 10, then the grammar to define a variable *sem* is: semaphore (10) *sem*.

Before using a semaphore variable, we need call function *sem_init* to initialize it. Function *sem_init* has three parameters, among which *sem* is the semaphore to be initialized; *value* is the initial resource count; *procNum* is the total number of processes using the resource. The function is defined as follows.

```
function sem_init( semaphore(n) *sem, int value, int procNum){
    frame(i) and (
        int i and i<==0 and empty;
        sem→value<==value and empty  //Init resource number
            sem→locked<== 0 and          //Init lock status
            sem→procNum<==procNum ; //Init process number
        while(i < n) {   //Init applying array and authorized array
            sem→curApp[i]:=0 and sem→runAuth[i]:=0;
            i:=i+1
        }
    )
};
```

Function *sem_acquire* corresponds to *wait* operation and allocate one unit of resource to the applier. The function has two parameters: *sem* is the resource

related semaphore; *id* is the identifier of the process applying for the resource. It firstly sets the applying flag and running authorisation flag of process *id* to 1 and 0 respectively, and then calls the function *_sem_lock* to lock the semaphore and mutual exclusively enters the critical section. Subsequently, the function minuses $sem \rightarrow value$ by 1 and checks the result. If $sem \rightarrow value < 0$, which means there is no resource left, then it adds the current process *id* to the tail of the blocked process queue *procQue*, frees the semaphore lock(i.e. $sem \rightarrow locked = 0$) and waits for other process to wake up process *id*(i.e. $await(sem \rightarrow runAuth[id] = 1)$. Otherwise, it frees the semaphore lock and uses the resource directly. The definition of function *sem_acquire* is as follows.

```
function sem_acquire( semaphore(n) *sem, int id) {
    sem→curApp[id]:= 1 and        //Set applying flag
        sem→runAuth[id]:= 0;      //Set running authorisation flag
    _sem_lock(sem, id);           //Lock the semaphore
    sem→value:=sem→value-1;
    if(sem→value< 0) then{
        sem→procQue.addtail(id);  //Add to waiting queue
        sem→locked=0;             //Free the semaphore lock
        await(sem→runAuth[id]= 1)
    }else {
        sem→locked=0              //Free the semaphore lock
    }
};
```

Function *sem_release* corresponds to *signal* operation and releases a resource. The function firstly sets the applying flag of process *id* to 1, and then calls the function *_sem_lock* to mutual exclusively enters the critical section. Subsequently, the function increases $sem \rightarrow value$ by 1 and checks the result. If $sem \rightarrow value <= 0$, which means there exists some processes blocked in the queue *procQue*, then it removes the first one from the head of the queue (i.e. $sem \rightarrow procQue.removehead(\&idWake)$ and wakes up it (i.e. $sem \rightarrow runAuth[idWake] := 1$). Finally, it frees the semaphore lock. The definition of function *sem_release* is as follows.

```
function sem_release( semaphore(n) *sem, int id){
    frame(idWake) and (       //ID of the process to be wake up
        int idWake and idWake<== 0 and empty;
        sem→curApp[id]:= 1;   //Set the applying flag
        _sem_lock(sem, id);
        sem→value:=sem→value+1;
        if(sem→value<= 0) then{
            sem→procQue.removehead(&idWake);
            sem→runAuth[idWake]:= 1;
        };
    sem→lock=0                //Free the semaphore lock
)
};
```

Function *_sem_lock* is used to identify which process can enter the critical section of the semaphore. The function takes the FCFS (First Come, First Service) strategy to select a process. To this end, it calls function *_sem_select* with the **external call** to select a process. If current process *id* is selected, it locks the semaphore (i.e. *sem→locked* <== 1) and removes the applying flag of process *id* (i.e. *sem→curApp*[*id*] := 0). Otherwise, the function increases the *sem → curApp*[*id*] by 1 to promote its priority to enter the critical section. The definition of function *_sem_lock* is as follows.

```
function _sem_lock(semaphore(n) *sem, int id){
    frame(idSel) and (
        int idSel and idsel<== −1 and empty;
        while(idSel ! = id) {
            ext _sem_select(sem, &idSel);
            if(idSel=id) then{
                sem→locked<== 1 and empty; //Lock the semaphore
                sem→curApp[id]:= 0          //Remove applying flag
            }else{
                sem→curApp[id]:= sem→curApp[id]+1
            }
        }
    )
};
```

Function *_sem_select* is used to select the process with the maximum value in the semaphore's applying array *curApp*. If the semaphore is locked, then no process is selected (i.e. ∗*idSel* := −1). Otherwise, the function traverses array *curApp* and finds the maximum one. The definition of *_sem_wait* is as follows.

```
function _sem_select( semaphore(n) *sem, int *idSel){
    frame(i, max) and (
        int i and i<== 0 and empty;
        int max and max<== 0 and skip;
        if(sem→locked=1) then {
            *idSel := −1
        } else {
            while(i<sem→procNum ) {
                if( sem→curApp[i]> sem→curApp[max]) then{
                    max := i
                };
                i := i+1
            };
        *idSel := max
    }
)
};
```

4.1 Application of Semaphore

In following, we give an example to illustrate how to employ the technique of semaphore to solve the complex Producer-consumer problem [14]. Without loss of generality, we assume the size of the buffer is 10, and the number of producers and consumers both be 2. In such a problem, there exist 3 kinds of critical resources, i.e., the buffer space, the product and the buffer itself. For each critical resource, we define a semaphore variable, namely *semSpace*, *semProd* and $semBuf$, and their initial resource numbers are $10, 0, 1$ respectively. The full MSVL program is given in Example 2.

Example 2. Solve Producer-consumer problem using semaphore

```
function Producer( semaphore(4) *sSpace, semaphore(4) *sProd,
                   semaphore(4) *sBuf, list(int) *buf, int id ){
    while(true) {
        sem_acquire(sSpace, id);    //Acquire a buffer space
        sem_acquire(sBuf, id);      //Acquire buffer
        buf→addtail(100);           //Imitate putting product into the buffer
        sem_release(sBuf, id);      //Release buffer
        sem_release(sProd, id);     //Release a product
    }
};
function Consumer( semaphore(4) *sSpace, semaphore(4) *sProd,
                   semaphore(4) *sBuf, list(int) *buf, int id ) {
    while(true) {
        sem_acquire(sProd, id);     //Acquire a product
        sem_acquire(sBuf, id);      //Acquire buffer
        buf→removehead();           //Imitate getting product from the buffer
        sem_release(sBuf, id);      //Release buffer
        sem_release(sSpace, id);    //Release a buffer space
    }
};
function main(){
    frame(semSpace, semProd, semBuf, buffer) and (
        semaphore(4) semSpace and sem_init(&semSpace, 10, 4);
        semaphore(4) semProd and sem_init(&semProd, 0, 4);
        semaphore(4) semBuf and sem_init(&semBuf, 1, 4);
        list(int) buffer;
        Producer(&semSpace, &semProd, &semBuf, &buffer, 0 )
            || Producer(&semSpace, &semProd, &semBuf, &buffer, 1 )
            || Consumer(&semSpace, &semProd, &semBuf, &buffer, 2 )
            || Consumer(&semSpace, &semProd, &semBuf, &buffer, 3 )
    )
};
main()
```

5 Conclusion

In this paper, we extend MSVL by introducing the technique of semaphore. The new semaphore type is defined in MSVL, and three operators *sem_init*, *sem_acquire* and *sem_release* are formalized to initialize a semaphore variable, acquire a resource and release a resource respectively. With the support of semaphore, MSVL can easily solve the synchronization, communication, and mutual exclusion problems between currently processes. In the future, we will apply MSVL to model, simulate and verify more complex concurrent and distributed systems, e.g. Operating System and Service Oriented System.

References

1. Duan, Z., Yang, X., Koutny, M.: Framed temporal logic programming. Sci. Comput. Program. **70**(1), 31–61 (2008)
2. Duan, Z.: Temporal Logic and Temporal Logic Programming. Science Press, Beijing (2005)
3. Duan, Z., Tian, C., Zhang, L.: A decision procedure for propositional projection temporal logic with infinite models. Acta Inf. **45**(1), 43–78 (2008)
4. Wang, M., Duan, Z., Tian, C.: Simulation and verification of the virtual memory management system with MSVL. In: Proceedings of the 2014 IEEE 18th International Conference on Computer Supported Cooperative Work in Design (CSCWD), pp. 360–365, May 2014
5. Yu, Y., Duan, Z., Tian, C., Yang, M.: Model checking C programs with MSVL. In: Liu, S. (ed.) SOFL 2012. LNCS, vol. 7787, pp. 87–103. Springer, Heidelberg (2013)
6. Ma, Q., Duan, Z., Zhang, N., Wang, X.: Verification of distributed systems with the axiomatic system of MSVL. Formal Asp. Comput. **27**(1), 103–131 (2015)
7. Tian, C., Duan, Z.: Expressiveness of propositional projection temporal logic with star. Theor. Comput. Sci. **412**(18), 1729–1744 (2011)
8. Duan, Z., Tian, C.: A unified model checking approach with projection temporal logic. In: Liu, S., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 167–186. Springer, Heidelberg (2008)
9. Milner, R.: Communication and Concurrency. Prentice Hall, London (1989)
10. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall, London (1985)
11. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997)
12. Dijkstra, E.W.: Over de sequentialiteit van procesbeschrijvingen (EWD-35). E.W. Dijkstra Archive. Center for American History, University of Texas at Austin
13. Zhang, N., Duan, Z., Tian, C.: Extending MSVL with function calls. In: Merz, S., Pang, J. (eds.) ICFEM 2014. LNCS, vol. 8829, pp. 446–458. Springer, Heidelberg (2014)
14. Arpaci-Dusseau, R.H.: Operating Systems: Three Easy Pieces [Chapter: Condition Variables]. Arpaci-Dusseau Books (2014)