Extending MSVL with Function Calls[⋆]

Nan Zhang, Zhenhua Duan[⋆⋆], and Cong Tian

Institute of Computing Theory and Technology,
and ISN Laboratory Xidian University, Xi'an 710071, China
zhhduan@mail.xidian.edu.cn, nanzhang@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 easier to use, we extend MSVL with function calls in this paper. To do so, an approach for function calls similar as in imperative programming languages is presented. Further, the semantics of expressions is redefined and the semantics of new added function call statements is formalized. Moreover, an example is given to illustrate how to use function calls in practice with MSVL.

Keywords: Temporal Logic Programming, Projection, Function Call, Modeling, Simulation, Verification.

1 Introduction

Modeling, Simulation and Verification Language (MSVL) [1] is a useful formalism for specification and verification of concurrent and real time systems [2,4,6,7,11]. It contains common statements used in most of imperative programming languages (e.g. C, Java) such as assignment, sequential ($\varphi_1; \varphi_2$), branch ($\texttt{if}\ b\ \texttt{then}\ \varphi_1\ \texttt{else}\ \varphi_2$) and iteration ($\texttt{while}\ b\ \texttt{do}\ \varphi$) statements but also parallel and concurrent statements such as conjunct ($\varphi_1\ \texttt{and}\ \varphi_2$), parallel ($\varphi_1 \| \varphi_2$) and projection ($(\varphi_1, \ldots, \varphi_m)\ \mathsf{prj}\ \varphi$) statements. The projection construct enables us to model a system in two time scales: with the fine-grained time interval, $\varphi_1, \ldots, \varphi_m$ are sequentially executed whereas with the coarse-grained interval called projected interval consisting of the executing end points of each program φ_i, φ is paralleled executed to monitor or control all or some of φ_i. This construct is particularly useful for modeling and simulating scheduling and real time systems [3,9,11]. Further, a Cylinder Computation Model (CCM) is proposed and included into MSVL [10,11], which can be used to describe and reason about multi-core parallel programs. Moreover, asynchronous communication mechanism has also been implemented in MSVL [5] which can be employed to model and verify distributed systems. To make MSVL more practical and useful, multi-types such as `integer`, `float`, `char`, `string`, `pointer` and `struct` etc. [8] have been recently formalized and implemented. Therefore, multi-typed values, functions and predicates concerning the extended data domain can be defined. However, functions calls as a kind of useful building

[⋆] The research is supported by the National Program on Key Basic Research Project of China (973 Program) Grant No.2010CB328102, National Natural Science Foundation of China under Grant No. 61133001, 61202038, 61272117, 61272118, 61322202 and 91218301.

[⋆⋆] Corresponding author.

block have not been formalized and implemented in MSVL yet so far. So, we are motivated to formalize a scheme to realize function calls based on multi-types.

The contributions of the paper are twofold: (1) Function definitions are formalized. With our scheme, a programmer is allowed not only to define new functions themselves but also to directly employ C library functions. Function definitions can be classified into four categories in terms of arguments and return value: with arguments and return value, with arguments but no return value, without arguments but with return value, with no arguments or return value. (2) Two kinds of function calls, black-box calling (short for b-call or ext-call) and white-box calling (short for w-call), are formalized. If we concern only the return value of a function but do not care about the interval over which the function is executed, a function ext-call should be employed. Most of function ext-calls are used in expressions. On the other hand, if we concern both the return value and the executed interval of a function, a function w-call should be used.

The rest of the paper is organized as follows: PTL and MSVL are briefly reviewed in the next section. Then, functions calls scheme is introduced in section 3, including the formalization of function definitions and function calls. In section 4, an example is given to illustrate how to program and call functions in MSVL. Finally, conclusions are drawn in section 5.

2 Preliminaries

2.1 PTL

In this section, the syntax and semantics of the underlying logic, Projection Temporal Logic (PTL), are briefly introduced. For more detail, please refer to paper [1].

Syntax Let $\mathbb{P}$ be a countable set of propositions, and $\mathbb{V}$ a countable set of typed variables consisting of static and dynamic variables. It is assumed that the value of a static variable remains the same over an interval (defined later) whereas a dynamic variable can have different values at different states. $\mathbb{B}$ represents the boolean domain $\{tt, ff\}$, $\mathbb{D}$ denotes all data needed by us including integers, lists, sets etc. $\mathbb{Z}$ denotes all integers, $\mathbb{N}_0$ stands for non-negative integers and $\mathbb{N}$ denotes positive integers. Terms e and formulas ϕ are inductively defined as follows:

$$\begin{aligned} e &::= u \mid \bigcirc e \mid \ominus e \mid f(e_1, \ldots, e_n) \\ \phi &::= q \mid e_1 = e_2 \mid P(e_1, \ldots, e_n) \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \exists x : \phi \mid \bigcirc \phi \mid (\phi_1, \ldots, \phi_m) \,\mathsf{prj}\, \phi \end{aligned}$$

where $u,\ x \in \mathbb{V}$ and $q \in \mathbb{P}$. A formula (term) is called a state formula (term) if it contains no temporal operators, i.e. $\bigcirc$, $\ominus$, prj, otherwise it is a temporal formula (term).

Semantics A state s over $\mathbb{V} \cup \mathbb{P}$ is defined to be a pair (I_v, I_p) of state interpretations I_v and I_p. I_v assigns each variable $u \in \mathbb{V}$ a value in $\mathbb{D}$ or nil (undefined) and the total domain is denoted by $\mathbb{D}' = \mathbb{D} \cup \{nil\}$, whereas I_p assigns each proposition $q \in \mathbb{P}$ a truth value in $\mathbb{B}$. $s[u]$ denotes the value of u at state s.

An interval σ is a non-empty sequence of states, which can be finite or infinite. The length, $|\sigma|$, of σ is ω if σ is infinite, and the number of states minus 1 if σ is finite. We extend the set $\mathbb{N}_0$ of non-negative integers to include ω, denoted by $\mathbb{N}_\omega = \mathbb{N}_0 \cup \{\omega\}$ and extend the comparison operators, $=$, $<$, $\leq$, to $\mathbb{N}_\omega$ by considering $\omega = \omega$, and for all $i \in \mathbb{N}_0$, $i < \omega$. Furthermore, we define $\preceq$ as $\leq -\{(\omega, \omega)\}$. For conciseness of presentation, $\langle s_0, \ldots, s_{|\sigma|}\rangle$ is denoted by σ, where $s_{|\sigma|}$ is undefined if σ is infinite. The concatenation of a finite σ with another interval (or empty string) $\sigma^{'}$ is denoted by $\sigma \cdot \sigma^{'}$ (not sharing any states). Let $\sigma = \langle s_0, s_1, \ldots, s_{|\sigma|}\rangle$ 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) $\sigma \downarrow (r_1, \ldots, r_h) = \langle s_{t_1}, s_{t_2}, \ldots, s_{t_l}\rangle$ where $t_1, \ldots, t_l$ are obtained from $r_1, \ldots, r_h$ by deleting all duplicates. That is, $t_1, \ldots, t_l$ is the longest strictly increasing subsequence of $r_1, \ldots, r_h$. For instance, $\langle s_0, s_1, s_2, s_3, s_4\rangle \downarrow (0, 0, 2, 2, 2, 3) = \langle s_0, s_2, s_3\rangle$. We also need to generalize the notation of $\sigma \downarrow (r_1, \ldots, r_h)$ to allow r_i to be ω. For an interval $\sigma = \langle s_0, s_1, \ldots, s_{|\sigma|}\rangle$ and $0 \leq r_1 \leq r_2 \leq \ldots \leq r_h \leq |\sigma|$ ($r_i \in \mathbb{N}_\omega$), we define $\sigma \downarrow (r_1, \ldots, r_h, \omega) = \sigma \downarrow (r_1, \ldots, r_h)$. To evaluate the existential quantification, an equivalence relation is required and given below. We use I_v^k and I_p^k to denote the state interpretations at state s_k.

Definition 1 (x- equivalence) Two intervals, σ and σ', are x-equivalent, denoted by $\sigma' \overset{x}{=} \sigma$, if $|\sigma| = |\sigma'|$, $I_v^h[y] = {I'}_v^h[y]$ for all $y \in \mathbb{V} - \{x\}$, and $I_p^h[q] = {I'}_p^h[q]$ for all $q \in \mathbb{P}$ $(0 \leq h \preceq |\sigma|)$.

An interpretation is a quadruple $\mathcal{I} = (\sigma, i, k, j)$, where σ is an interval, $i, k \in \mathbb{N}_0$, and $j \in \mathbb{N}_\omega$ such that $0 \leq i \leq k \preceq j \leq |\sigma|$. We use the notation (σ, i, k, j) to indicate that some formula ϕ or term e is interpreted over the subinterval $\langle s_i, \ldots, s_j\rangle$ of σ with the current state being s_k. For every term e, the evaluation of e relative to interpretation $\mathcal{I} = (\sigma, i, k, j)$, denoted by $\mathcal{I}[e]$, is defined by induction on terms as follows:

$$1.\ \mathcal{I}[u] = \begin{cases} s_k[u] = I_v^k[u] = I_v^i[u] & \text{if } u \text{ is a static variable.} \\ s_k[u] = I_v^k[u] & \text{if } u \text{ is a dynamic variable.} \end{cases}$$

$$2.\ \mathcal{I}[\bigcirc e] = \begin{cases} (\sigma, i, k+1, j)[e] & \text{if } k < j \\ nil & \text{otherwise} \end{cases}$$

$$3.\ \mathcal{I}[\ominus e] = \begin{cases} (\sigma, i, k-1, j)[e] & \text{if } i < k \\ nil & \text{otherwise} \end{cases}$$

$$4.\ \mathcal{I}[f(e_1, \ldots, e_n)] = \begin{cases} nil & \text{if } \mathcal{I}[e_h] = nil, \text{ for some } h \in \{1, \ldots, n\} \\ \mathcal{I}[f](\mathcal{I}[e_1], \ldots, \mathcal{I}[e_n]) & \text{otherwise} \end{cases}$$

The meaning of formulas is given by the satisfaction relation, $\models$, which is inductively defined as follows:

1. $\mathcal{I} \models q$ iff $I_p^k[q] = tt$, for any given proposition q.
2. $\mathcal{I} \models P(e_1, \ldots, e_n)$ iff P is a primitive predicate other than $=$ and, for all h, $1 \leq h \leq n, \mathcal{I}[e_h] \neq nil$ and $P(\mathcal{I}[e_1], \ldots, \mathcal{I}[e_n]) = tt$.
3. $\mathcal{I} \models e_1 = e_2$ iff e_1 and e_2 are terms and $\mathcal{I}[e_1] = \mathcal{I}[e_2]$.

4. $\mathcal{I} \models \neg\phi$ iff $\mathcal{I} \not\models \phi$.
5. $\mathcal{I} \models \bigcirc\phi$ iff $k < j$ and $(\sigma, i, k+1, j) \models \phi$.
6. $\mathcal{I} \models \phi_1 \wedge \phi_2$ iff $\mathcal{I} \models \phi_1$ and $\mathcal{I} \models \phi_2$.
7. $\mathcal{I} \models \exists x : \phi$ iff there exists an interval σ' such that $\sigma'_{(i..j)} \overset{x}{=} \sigma_{(i..j)}$ and $(\sigma', i, k, j) \models \phi$.
8. $\mathcal{I} \models (\phi_1, \ldots, \phi_m)$ prj ϕ iff there exist integers $k = r_0 \leq \cdots \leq r_{m-1} \preceq r_m \leq j$ such that for all $1 \leq l \leq m$, $(\sigma, i, r_{l-1}, r_l) \models \phi_l$, and $(\sigma', 0, 0, |\sigma'|) \models \phi$ for one of the following σ' :
   (a) $r_m < j$ and $\sigma' = \sigma \downarrow (r_0, \ldots, r_m) \cdot \sigma_{(r_m+1..j)}$, or
   (b) $r_m = j$ and $\sigma' = \sigma \downarrow (r_0, \ldots, r_h)$ for some $0 \leq h \leq m$.

ABBREVIATION The abbreviations *true*, *false*, $\wedge$, $\rightarrow$ and $\leftrightarrow$ are defined as usual. In particular, $true \overset{\text{def}}{=} \phi \vee \neg\phi$ and $false \overset{\text{def}}{=} \phi \wedge \neg\phi$ for any formula ϕ. The derived formulas are given as follows, where $n \in \mathbb{N}_0$.

A1	$more \overset{\text{def}}{=} \bigcirc true$	A2	$\varepsilon \overset{\text{def}}{=} \neg \bigcirc true$
A3	$\bigcirc^0 \phi \overset{\text{def}}{=} \phi$	A4	$\bigcirc^{n+1}\phi \overset{\text{def}}{=} \bigcirc(\bigcirc^n \phi)$
A5	$\phi_1 ; \phi_2 \overset{\text{def}}{=} (\phi_1, \phi_2)\ prj\ \varepsilon$	A6	$\Diamond\phi \overset{\text{def}}{=} true ; \phi$
A7	$\Box\phi \overset{\text{def}}{=} \neg\Diamond\neg\phi$	A8	$\odot\phi \overset{\text{def}}{=} \varepsilon \vee \bigcirc\phi$
A9	$\phi^0 \overset{\text{def}}{=} \varepsilon$	A10	$\phi^{n+1} \overset{\text{def}}{=} \phi^n ; \phi$
A11	$len(n) \overset{\text{def}}{=} \bigcirc^n \varepsilon$	A12	$skip \overset{\text{def}}{=} \mathsf{len}(1)$
A13	$fin(p) \overset{\text{def}}{=} \Box(\varepsilon \rightarrow p)$	A14	$inf \overset{\text{def}}{=} \neg\Diamond\varepsilon$
A15	$keep(p) \overset{\text{def}}{=} \Box(more \rightarrow p)$	A16	$halt(p) \overset{\text{def}}{=} \Box(\varepsilon \leftrightarrow p)$

2.2 MSVL

Modeling, Simulation and Verification Language (MSVL) is an executable subset of PTL. The following is a snapshot of the simple kernel of MSVL. For more detail, please refer to paper [1]. With MSVL, expressions can be treated as terms and statements can be treated as formulas in PTL. The arithmetic and boolean expressions of MSVL can be inductively defined as follows:

$$\begin{aligned} e &::= n \mid x \mid \bigcirc x \mid \ominus x \mid e_0 + e_1 \mid e_0 - e_1 \mid e_0 * e_1 \mid e_0 \% e_1 \\ b &::= tt \mid ff \mid \neg b \mid b_0 \wedge b_1 \mid e_0 = e_1 \mid e_0 < e_1 \end{aligned}$$

where n is an integer and x is a static or dynamic variable. One may refer to the value of a variable at the previous state or the next state. The statements of MSVL can be inductively defined in the following table:

	Name	Symbol φ	PTL Definition $\mathcal{F}(\varphi)$
1	Termination	$empty$	ε
2	Assignment	$x := e$	$\bigcirc x = e \wedge \bigcirc p_x \wedge skip$
3	Positive Immediate Assignment	$x <== e$	$x = e \wedge p_x$
4	State Frame	$\mathsf{lbf}(x)$	$\neg\mathsf{af}(x) \rightarrow \exists\, b : (\ominus x = b \wedge x = b)$

5	Interval Frame	$\mathsf{frame}(x)$	$\Box(more \to \bigcirc\mathcal{F}(\mathsf{lbf}(x)))$
6	Next	$\texttt{next}\ \varphi$	$\bigcirc\mathcal{F}(\varphi)$
7	Always	$\texttt{always}\ \varphi$	$\Box\mathcal{F}(\varphi)$
8	Conditional	$\texttt{if}\ b\ \texttt{then}\ \varphi_0\ \texttt{else}\ \varphi_1$	$(b \to \mathcal{F}(\varphi_0)) \wedge (\neg b \to \mathcal{F}(\varphi_1))$
9	Existential Quantification	$\texttt{exist}\ x : \varphi$	$\exists\, x : \mathcal{F}(\varphi)$
10	Sequential	$\varphi_0; \varphi_1$	$\mathcal{F}(\varphi_0); \mathcal{F}(\varphi_1)$
11	Conjunction	$\varphi_0\ \texttt{and}\ \varphi_1$	$\mathcal{F}(\varphi_0) \wedge \mathcal{F}(\varphi_1)$
12	While	$\texttt{while}\ b\ \texttt{do}\ \varphi$	$(b \wedge \mathcal{F}(\varphi))^* \wedge \Box(\varepsilon \to \neg b)$
13	Selection	φ_0 or φ_1	$\mathcal{F}(\varphi_0) \vee \mathcal{F}(\varphi_1)$
14	Parallel	$\varphi_0 \| \varphi_1$	$\mathcal{F}(\varphi_0) \wedge (\mathcal{F}(\varphi_1); tt) \vee (\mathcal{F}(\varphi_0); tt) \wedge \mathcal{F}(\varphi_1)$
15	Projection	$(\varphi_1, \ldots, \varphi_m)\ \mathsf{prj}\ \varphi$	$(\mathcal{F}(\varphi_1), \ldots, \mathcal{F}(\varphi_m))\ \mathsf{prj}\ \mathcal{F}(\varphi)$
16	Interval Length	$\texttt{len}(n)$	$\bigcirc^n \varepsilon$
17	Synchronous Communication	$\mathsf{await}(c)$	$\mathcal{F}(\mathsf{frame}(x_1, \ldots, x_n)) \wedge \Box(\varepsilon \leftrightarrow c)$

MSVL supports structured programming and covers some basic control flow statements such as sequential statement, conditional statement, while-loop statement and so on. Further, MSVL also supports non-determinism and concurrent programming by including selection, conjunction and parallel statements. Moreover, a framing technique is introduced to improve the efficiency of programs and synchronize communication for parallel processes. In addition, MSVL has been extended in a variety of ways. For instance, multi-types have been recently formalized and implemented [8]. Hence, typed variables, typed functions and predicates over the extended data domain can be defined.

3 Introducing Function Calls into MSVL

We extend MSVL in this section by introducing and formalizing function definitions and calls, including the syntax and semantics. Since we permit the appearance of function calls with return values in expressions, the form and interpretation of MSVL expressions also need to be reconsidered.

3.1 Data Types

Like C programming language, MSVL provides a variety of data types. The fundamental types are unsigned characters (`char`), unsigned integers (`int`) and floating point numbers (`float`). In addition, there is a hierarchy of derived data types built with strings (`string`), lists (`list`), pointers (`pointer`), arrays (`array`), structures (`struct`) and unions (`union`). For more detail, please refer to paper [8].

3.2 Function Calls

There are two kinds of functions in MSVL: one is external functions, written in other programming languages such as C and Java and the other is user-defined functions written in MSVL.

General Principles. MSVL can only define functions and predicates. The so called functions in C are mixed cases of functions and predicates. Generally speaking, the following statements can be used to define state functions and predicates:

$$\textbf{define}\ \text{type}\ f(\text{type}_1\ x_1, ..., \text{type}_n\ x_n) \overset{\text{def}}{=} e$$
$$\textbf{define}\ P(\text{type}_1\ x_1, ..., \text{type}_n\ x_n) \overset{\text{def}}{=} \varphi$$

where $x_1, ..., x_n$ are typed state variables and e a typed expression while φ is a statement. Thus, f is defined as a typed n arity function while P is defined as an n arity predicate. A state function can be called by substituting arguments $e_1, ..., e_n$ for parameters $x_1, ..., x_n$ respectively within an expression while a predicate can be invocated in a similar way but as a statement. For example,

$$\textbf{define}\ \texttt{float}\ \mathit{max}(\texttt{float}\ x, \texttt{float}\ y) \overset{\text{def}}{=} \texttt{if}\ (x > y)\ \texttt{then}\ x\ \texttt{else}\ y$$

defines a state function *max* which can be used in an expression such as $9.5 + \mathit{max}(7.5, 8.5)$.

```
    define max(int a[ ], int lim, int x)
def= frame(temp, i) and (
    int temp := a[0];
    int i := 1;
    while (i ≤ lim − 1) do
      {(if a[i] > temp then temp := a[i]); i := i + 1};
    x := temp;
    )
```

The above defines a predicate *max* which chooses a maximum element from an array with length lim. To call it, we only need to replace all parameters by arguments and make a statement: $\mathit{max}(x[9, 8, 7, 1, 9, 2, 3, 6, 5], 9, y)$, which chooses the maximum from the array x and stores the result into the variable y.

(1) External function calls

If we permit an MSVL program to call an external functions written in C or Java such as C standard library functions, the situation turns to be complicated since we do not know the interval of the execution of an external function. Nevertheless if we do not care about the executed interval of an external function but concern only with its return value and output results, we could simplify the calling process. In fact, a standard definition of C functions is of the following form:

$$\textbf{return\_type}\ g(\textbf{in\_type}_1\ x_1, ..., \textbf{in\_type}_n\ x_n, \textbf{out\_type}_1\ y_1, ..., \textbf{out\_type}_m\ y_m)$$

where g is a function with $x_1, ..., x_n$ as its typed input parameters while $y_1, ..., y_m$ as its typed output parameters and return_type as the type of its return value. For example,

$$\texttt{int}\ \mathit{getline}(\texttt{int}\ lim,\ \texttt{char}\ s[\,])$$

is a C function which reads a character line into array $s[\,]$ with length limited by input parameter lim and returns the actual length of the string. In some circumstances, input or output parameters or return value or all of them can be omitted (denoted by void). In order to call this type of functions as a statement in an MSVL program, the C functions need to be slightly modified in C as shown below:

$$\texttt{void}\, g(\textbf{in\_type}_1\ x_1, ..., \textbf{in\_type}_n\ x_n, \textbf{out\_type}_1\ y_1, ..., \textbf{out\_type}_m\ y_m, \textbf{return\_type}\ RV[\,])$$

where we add an extra typed return parameter to the function. Note that the last "**return** val" statement in a C function now needs to be replaced by an assignment "$RV[0] = val$" statement in the function without changing other statements.

To call this kind of functions as a statement in an MSVL program without concerning the interval on which the function is executed, we make a new statement below:

$$\texttt{ext}\, g(\ e_1, ...,\ e_n,\ z_1, ...,\ z_m,\ R)$$

For example, *getline* function written in C can be re-written as follows:

$$\texttt{void}\, \textit{getline}(\texttt{int}\ lim, \texttt{char}\ s[\,], \texttt{int}\ rv[\,])$$

This new function can be directly called in an MSVL program as a statement:

$$\texttt{ext}\ \textit{getline}(10, x, l);$$

Of course, a C function without a return value can be directly called using the above form.

If an external function without output parameters but with a return value, since we only concern the return value of an external function, it can directly be called in an expression. For example, C function `int` *strlen*(`char` $s[\,]$), returning the length of a string s, can be employed in an expression in MSVL program:

$$\texttt{ext}\ \textit{strlen}(\text{“hello world!”}) + \texttt{ext}\ \textit{strlen}(\text{“Good morning!”}) \geq 10$$

(2) User-defined function calls

If an external function modifies memory units or program variables, it is required to redefine in MSVL and cannot be directly called from an MSVL program. For example, `void` $*$ *memcpy*(s, *ct*, n) is a standard C function which copies n characters from *ct* to s and returns s. When this function is used, a pointer pointing memory address could be returned. Therefore, it is not permitted to be called in an MSVL program. To use this kind of external functions, the only way is to redefine them in MSVL. User defined functions can be classified into four categories: (a) functions with arguments and return value; (b) functions with return value but without arguments; (c) functions with arguments but without return value; (d) functions with no arguments or return value. Generally, a user defined function is of the following form in MSVL:

$$\texttt{define}\, g(\textbf{in\_type}_1\ x_1, ..., \textbf{in\_type}_n\ x_n, \textbf{out\_type}_1\ y_1, ..., \textbf{out\_type}_m\ y_m, \textbf{return\_type}\ RV[\,])$$

where we add an extra typed return parameter to the function as output parameter. Note that the last "**return** val" statement in a C function now needs to be replaced by an assignment "$RV[0] := val$" statement in an MSVL function if we try to redefine the C function. In fact, it is really a predicate with two kinds of parameters: input and output parameters. To call this kind of functions in an MSVL program, it is simply to write the following statement with input arguments $e_1, ..., e_n$ and output arguments $z_1, ..., z_m$ and R:

$$g(\ e_1, ...,\ e_n,\ z_1, ...,\ z_m,\ R);$$

For example, *getline* function written in C can be re-written as follows:

$$\textit{getline}(\texttt{int}\ lim, \texttt{char}\ s[\,], \texttt{int}\ rv[\,]) \overset{\text{def}}{=} Q$$

where Q is defined in the MSVL program below:

```
/* getline: get line into $s$, store length into rv */
define getline(int $lim$, char $s[\,]$, int $rv[\,]$)
{
        frame($s[\,], lim, c, RValue$)
         and int $RValue <== 0$
         and char $c <==$ ext getchar( )
         and (
              while ($lim - 1 > 0$ and $c! = EOF$ and $c! = '\backslash n'$)
               {
                 $s[RValue] <== c$ and
                 $RValue := RValue + 1$ and
                 $lim := lim - 1$ and
                 $c :=$ ext getchar( )
               };
               if ($c = '\backslash n'$) then $s[RValue] <== c$ and $RValue := RValue + 1$;
               $s[RValue] <== '\backslash 0'$;
               $rv[0] := RValue$
             )
}
```

Now *getline* function can now be called as follows:

$$\texttt{ext}\ \textit{getline}(10, x, l);$$

3.3 Interpretation of Function Calls

There are two kinds of function calls in MSVL: (1) Black-box call or external call (short for b-calls or ext-calls): the interval over which the called function is executed is ignored. If a function neither changes any memory units nor uses any external variables whose scopes are not limited to the function, it can be called using black-box manner by the calling function. Such kind of function calls often appears in expressions. In other words, all the function calls appearing in expressions are external calls. (2) White-box call: the interval over which the called function is executed is inserted and concatenated with the main interval over which the calling function is executed. If a function uses some external variables, it should be called using white-box manner by the calling function.

(1) Interpretation of function calls in expressions

With expressions, function calls with black-box manner are only allowed. Since more data types have been included into MSVL, expressions should also be extended to cover more types. Thus expressions are inductively redefined as follows:

- Individual typed constants are basic expressions: a, b, c, $\ldots \in \mathbb{D}$ possibly with subscripts.
- Individual typed variables (static or dynamic) are basic expressions: u, v, x, y, z, $\ldots \in \mathbb{V}$ possibly with subscripts.
- Temporal operators: if e is an expression, then $\bigcirc e$ and $\odot e$ are expressions.
- Non-temporal operators: if op is an operator of arity n $(n > 0)$ in MSVL and $e_1, \ldots, e_n$ are expressions of types compatible with types of parameters of op, then $\mathtt{op}(e_1, \ldots, e_n)$ is an expression. The operators allowed in MSVL are given in the following list.

Multiplicative operators:	$*$, $/$, $\%$
Unary additive operators:	$+$, $-$
Binary additive operators:	$+$, $-$
Relational operators:	$=$, $!=$, $<$, $<=$, $>$, $>=$
Bitwise operators:	&, |, ^, <<, >>
Logical operators:	$\neg$, $\wedge$, $\vee$

- if h is a user defined state function of arity n $(n > 0)$ and $e_1, \ldots, e_n$ are expressions of types compatible with types of parameters of h, then $h(e_1, \ldots, e_n)$ is an expression;
- if f is a user defined function of arity n $(n > 0)$ and $e_1, \ldots, e_n$ are expressions of types compatible with types of parameters of f, then $\mathtt{ext}\ f(e_1, \ldots, e_n)$ is an expression;
- if g is an external function of arity n $(n > 0)$ and $e_1, \ldots, e_n$ are expressions of types compatible with types of parameters of g, then $\mathtt{ext}\ g(e_1, \ldots, e_n)$ is an expression;

For each expression e, the evaluation of e related to interpretation $\mathcal{I} = (\sigma, i, k, j)$, denoted by $\mathcal{I}[e]$, is redefined based on the semantics of PTL.

1. $\mathcal{I}[a] = a$ for each typed constant $a \in \mathbb{D}$.

2. $$\mathcal{I}[u] = \begin{cases} s_k[u] = I_v^k[u] = I_v^i[u] & \text{if } u \text{ is a static variable.} \\ s_k[u] = I_v^k[u] & \text{if } u \text{ is a dynamic variable.} \end{cases}$$

3. $$\mathcal{I}[\bigcirc e] = \begin{cases} (\sigma, i, k+1, j)[e] & \text{if } k < j. \\ nil & \text{otherwise.} \end{cases}$$

4. $\mathcal{I}[\bigcirc e] = \begin{cases} (\sigma, i, k-1, j)[e] & \text{if } i < k. \\ nil & \text{otherwise.} \end{cases}$

5. $\mathcal{I}[\texttt{op}(e_1, \ldots, e_n)] = \begin{cases} nil & \text{if } \mathcal{I}[e_i] = nil \text{ for some } i \in \{1, \ldots, n\}. \\ \mathcal{I}[\texttt{op}](\mathcal{I}[e_1], \ldots, \mathcal{I}[e_n]) & \text{otherwise.} \end{cases}$

op is an operator.

6. $\mathcal{I}[h(e_1, \ldots, e_n)] = \begin{cases} nil & \text{if } \mathcal{I}[e_i] = nil \text{ for some } i \in \{1, \ldots, n\}. \\ \mathcal{I}[h](\mathcal{I}[e_1], \ldots, \mathcal{I}[e_n]) & \text{otherwise.} \end{cases}$

h is a state function.

7. $\mathcal{I}[\texttt{ext} f(e_1, \ldots, e_n)] = \begin{cases} nil & \text{if } \mathcal{I}[e_i] = nil \text{ for some } i \in \{1, \ldots, n\}. \\ \mathcal{I}[f](\mathcal{I}[e_1], \ldots, \mathcal{I}[e_n]) & \text{otherwise.} \end{cases}$

f is a non-state function.

8. $\mathcal{I}[\texttt{ext } g(e_1, \ldots, e_n)] = \begin{cases} nil & \text{if } \mathcal{I}[e_i] = nil \text{ for some } i \in \{1, \ldots, n\}. \\ \mathcal{I}[g](\mathcal{I}[e_1], \ldots, \mathcal{I}[e_n]) & \text{otherwise.} \end{cases}$

g is an external function.

(2) Interpretation of function calls in statements

For the new statement ext $g(e_1, ..., e_n, z_1, ..., z_m, R)$, its interpretation is given as follows: Let $\sigma =< s_0, ..., s_k, ..., s_{|\sigma|} >$ be an interval, and $\mathcal{I} = (\sigma, i, k, j)$ be the interpretation, $Q \overset{\text{def}}{=} (g(e_1, ..., e_n, z_1, ..., z_m, R) \wedge \exists b_1 \cdots \exists b_n \exists r : \bigwedge_{i=1}^{m} fin(z_i = b_i) \wedge fin(R = r))$ prj $(z_i := b_i \wedge R := r)$. Thus,

$\mathcal{I} \models \texttt{ext } g(e_1, ..., e_n, z_1, ..., z_m, R)$ iff $j = k + 1$ and there exists an interval $\sigma'' =< s_k, s''_1, ..., s_{k+1} >$ such that $\sigma' = \sigma_{(0..k)} \cdot \sigma''$ and $(\sigma', i, k, |\sigma'|) \models Q$.

The model of Q is illustrated in Fig.1.

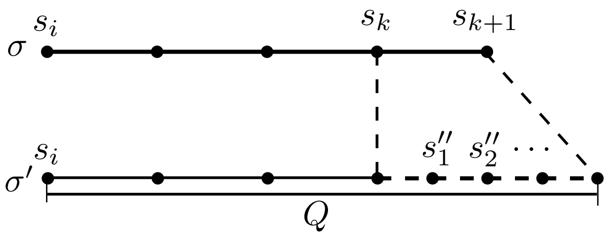

Fig. 1. Interpretation of external function calls

4 Example

In this section, we write a MSVL program to print each line of its input that contains a particular "pattern" of characters. For example, searching for the pattern of letters "ould" in the set of lines

Ah Love! could you and I with Fate conspire
To grasp this sorry Scheme of Things entire,
Would not we shatter it to bits – and then
Re-mould it nearer to the Heart's Desire!

will produce the output

Ah Love! could you and I with Fate conspire
Would not we shatter it to bits – and then
Re-mould it nearer to the Heart's Desire!

The MSVL program calls two functions: *getline* and *strindex*, which are defined in C before the main program. The *getline* function fetches the next line of input, stores it into s and stores the length of the line into RV. The *strindex* function records the position or index in the string s where the string t begins, or -1 if s doesn't contain t, and stores the result into RV.

C Program

```
/* getline: get line into s, store length into RV*/
void getline(int lim, char s[ ], int RV[ ])
{
        int c, i;
        i=0;
        while(−−lim>0 && (c=getchar( ))!=EOF && c!='\n')
                s[i++]=c;
        if(c=='\n')
                s[i++]=c;
        s[i]='\0';
        RV[0]=i;
}

/* strindex: store index of t in s into RV, -1 if none */
void strindex(char s[ ], char t[ ], int RV[ ])
{
        int i, j, k, r=0;
        for (i=0; s[i]!='\0'&& r=0;i++) {
                for (j=i, k=0; t[k]!='\0' && s[j]==t[k]; j++, k++)
                        ;
```

```
            if (k>0 && t[k]=='\0')
                  { r=1; RV[0]=i};
      }
      RV=-1;
}
```

MSVL Program

```
/* main program: find all lines matching pattern */
frame(MAX, pattern[ ], line[MAX], l[ ], in[ ], found, length, index)
 and int MAX <== 1000 and char pattern[ ] <== "ould"
 and int found <== 0 and int length and int index
 and (
      ext getline(MAX, line, l);
      length := l[0];
      while (length > 0)
        {
          ext strindex(line, pattern, in);
          index := in[0];
          if (index >= 0) then (printf("%s", line); found := found + 1);
          ext getline(MAX, line, l);
          length := l[0];
        }
    )
```

5 Conclusion

In this paper, MSVL is extended by means of formalizing function definitions and function calls. Functions with return value and no external variables are used to extend expressions. The function calls appear in expressions are called black-box (or external) calls. Functions with external variables can be called with the white-box manner as individual statement. With black-box calling, the intermediate execution detail of functions is ignored and only the return value is concerned. In white-box calling, the interval over which the function is interpreted makes up a part of the interval over which the whole program is interpreted. The function calls approach presented in this paper has been implemented in the interpreter of MSVL. In the future, we will apply MSVL to model, simulate and verify more practical applications.

References

1. Duan, Z.: Temporal logic and temporal logic programming. Science Press, Beijing (2005)
2. Han, M., Duan, Z., Wang, X.: Time constraints with temporal logic programming. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 266–282. Springer, Heidelberg (2012)
3. Liu, C., Layland, J.: Scheduling algorithm for multiprogramming in a hard real-time environment. Journal of the ACM 20(1), 46–61 (1973)
4. Manna, Z., Pnueli, A.: The temporal logic of reactive and concurrent system. Springer, New York (1992)
5. Mo, D., Wang, X., Duan, Z.: Asynchronous communication in MSVL. In: Qin, S., Qiu, Z. (eds.) ICFEM 2011. LNCS, vol. 6991, pp. 82–97. Springer, Heidelberg (2011)
6. Pnueli, A.: The temporal logic of programs. In: Proceedings of the 18th Annual Symposium on the Foundations of Computer Science, pp. 46–57. IEEE Computer Society, Providence (1977)
7. Queille, J., Sifakis, J.: Specification and verification of concurrent systems in cesar. In: Dezani-Ciancaglini, M., Montanari, U. (eds.) Proceedings of the 5th Colloquium on International Symposium in Programming. LNCS, vol. 137, pp. 337–351. Springer, Springer (1982)
8. Wang, X., Duan, Z., Zhao, L.: Formalizing and implementing types in msvl, pp. 62–75 (2013)
9. Zhan, N.: An intuitive formal proof for deadline driven scheduler. Journal of Computer Science and Technology 16(2), 146–158 (2001)
10. Zhang, N., Duan, Z., Tian, C.: A cylinder computation model for many-core parallel computing. Theoretical Computer Science 497, 68–83 (2013)
11. Zhang, N., Duan, Z., Tian, C., Du, D.: A formal proof of the deadline driven scheduler in pptl axiomatic system. Theoretical Computer Science (2014)