Formalizing and Implementing Types in MSVL

Xiaobing Wang, Zhenhua Duan(✉), and Liang Zhao

Institute of Computing Theory and Technology and ISN Laboratory,
Xidian University, Xian 710071, People's Republic of China
xbwang@mail.xidian.edu.cn, zhenhua_duan@126.com, lzhao@xidian.edu.cn

Abstract. This paper investigates techniques for formalizing and implementing types in the temporal logic programming language MSVL, which is an executable subset of Projection Temporal Logic. To this end, the data domain of MSVL is extended to include typed values, and then typed functions and predicates concerning the extended data domain are defined. Based on these definitions, the statement for type declaration of program variables is formalized. The implementation mechanisms of the type declaration statement in the MSVL interpreter are also discussed, which is based on the notion of normal form of MSVL programs. To illustrate how to program with types, an example of in-place reversing an integer list is given.

Keywords: Type · Temporal logic programming · MSVL · Projection Temporal Logic

1 Introduction

In many formal verification fields ranging from digit circuit design to software engineering, temporal logics have been widely used as an efficient tool for describing and reasoning about properties of concurrent systems [1–3]. Projection Temporal Logic (PTL) extends Interval Temporal Logic (ITL) and is widely applied to system specification and verification [4]. In most cases the system modeling techniques have nothing to do with temporal logics while the desired properties are described by temporal logic formulas, thus verification has suffered from a defect that different formal methods have different denotations and semantics. To improve this situation, one way is to use the same language for modeling systems and describing properties. Modeling, Simulation and Verification Language (MSVL) is a temporal logic programming language developed as an executable subset of PTL, where concurrent systems can be modeled, simulated and verified [5]. In this method, a concurrent system is modeled by an MSVL program while properties of this system are specified by Propositional Projection Temporal Logic (PPTL) formulas [6]. By model checking within the same temporal logic

This research is supported by the NSFC Grant Nos. 61272118, 61272117, 61133001, 61202038, 61373043, 973 Program Grant No. 2010CB328102, and the Fundamental Research Funds for the Central Universities Nos. K5051203014, K5051303020.

S. Liu and Z. Duan (Eds.): SOFL+MSVL 2013, LNCS 8332, pp. 62–75, 2014.
DOI: 10.1007/978-3-319-04915-1_5,

framework, whether or not a concurrent system satisfies the desired properties can be verified.

Introducing types into a programming languages is important, which enables us to write more sound and practical programs. As we known, most conventional programming languages such as C, C++ and Java have their own data types, including integer, array, list, etc. However, most temporal logic programming languages, e.g. MSVL, Tempura [7], XYZ/E [8], TLA [9] and METATEM [10], have not implemented types yet. To bridge the gap between temporal logic programming and conventional programming, we are motivated to investigate techniques for formalzing and implementing types in MSVL.

In [11], there are two basic built-in types, integer and bool, which can be given pure set-theoretic definitions in Tempura. Further types can be built from these basic types by means of the $\times$ operator and the power set operator. Also, they defined a statement $type(x, T)$ to introduce a variable x of a type T. But so far as we know, no further formalization and implementation details have been published to clarify a proper way to implement types in Tempura, neither does the Tempura interpreter consider the execution of type statements.

The main contributions of this paper are as follows. (1) The data domain D of MSVL is formalized to describe types including integer, float, character, array, list, etc. (2) Typed functions and predicates, and the type declaration statement are defined. (3) The normal form for those statements are given and they are implemented in the MSVL interpreter. With these contributions, the language MSVL can be used to model, simulate and verify typed programs.

The rest of the paper is organized as follows. Section 2 briefly introduces PTL and MSVL. In Sect. 3, the data domain, typed functions and predicates, and the type declaration statement are formalized. Then, Sect. 4 provides the MSVL interpreter implementation mechanisms based on the notion of the normal form. In Sect. 5, the MSVL interpreter is used to model and verify an in-place reversal of an integer list. Finally, conclusions are drawn in Sect. 6.

2 Preliminaries

2.1 Projection Temporal Logic

Let $\mathcal{P}$ be a countable set of propositions, and $\mathcal{V}$ be a countable set of typed static and dynamic variables. PTL terms e and formulas p are given by the following grammar [4].

$$
\begin{aligned}
e &::= x \mid \bigcirc e \mid \ominus e \mid f(e_1, \ldots, e_m) \\
p &::= r \mid e_1 = e_2 \mid P(e_1, \ldots, e_m) \mid \neg p \mid p_1 \wedge p_2 \mid \exists v : p \mid \\
&\quad \bigcirc p \mid (p_1, \ldots, p_m)\ \mathsf{prj}\ p
\end{aligned}
$$

where $r \in \mathcal{P}$ is a proposition, and $x \in \mathcal{V}$ is a dynamic or static variable. In $f(e_1, \ldots, e_m)$ and $P(e_1, \ldots, e_m)$, f is a function and P is a predicate. Each function and predicate has a fixed arity. A formula (term) is called a *state* formula

(term) if it does not contain any temporal operators (i.e. $\bigcirc$ or prj); otherwise it is a *temporal* formula (term).

A state s is a pair of assignments (I_v, I_p) where for each variable x defines $s[x] = I_v[x]$, and for each proposition r defines $s[r] = I_p[r]$. $I_v[x]$ is a value in the data domain $\mathcal{D}$ or *nil* which means "undefined", and $I_p[r]$ is a value in $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$. An interval $\sigma = \langle s_0, s_1, \ldots \rangle$ is a non-empty (possibly infinite) sequence of states. The length of σ, denoted by $|\sigma|$, is defined as ω if σ is infinite; otherwise it is the number of states in σ minus one. To have a uniform notation for both finite and infinite intervals, we will use extended integers as indices. That is, we consider the set $\mathbb{N}$ of natural numbers and ω, $\mathbb{N}_\omega = \mathbb{N} \cup \{\omega\}$, and extend the comparison operators, $=, <, \leq$, to $\mathbb{N}_\omega$ by considering $\omega = \omega$, and for all $i \in \mathbb{N}$, $i < \omega$. Moreover, we define $\preceq$ as $\leq -\{(\omega, \omega)\}$. With such a notation, $\sigma_{(i..j)}(0 \leq i \preceq j \leq |\sigma|)$ denotes the sub-interval $\langle s_i, \ldots, s_j \rangle$ and $\sigma(k)(0 \leq k \preceq |\sigma|)$ denotes $\langle s_k, \ldots, s_{|\sigma|} \rangle$. The concatenation of σ with another interval (or empty string) σ' is denoted by $\sigma \bullet \sigma'$. To define the semantics of the projection operator we need an auxiliary operator for intervals. Let $\sigma = \langle s_0, s_1, \ldots \rangle$ be an interval and $n_1, \ldots, n_h$ be integers $(h \geq 1)$ such that $0 \leq n_1 \leq n_2 \leq \ldots \leq n_h \preceq |\sigma|$. The projection of σ onto $n_1, \ldots, n_h$ is the interval (called projected interval), $\sigma \downarrow (n_1, \ldots, n_h) = \langle s_{m_1}, s_{m_2}, \ldots, s_{m_l} \rangle$, where $m_1, \ldots, m_l$ is obtained from $n_1, \ldots, n_h$ by deleting all duplicates. For example,

$$\langle s_0, s_1, s_2, s_3, s_4 \rangle \downarrow (0, 0, 2, 2, 2, 3) = \langle s_0, s_2, s_3 \rangle$$

An interpretation for a PTL term or formula is a tuple $\mathcal{I} = (\sigma, i, k, j)$, where $\sigma = \langle s_0, s_1, \ldots \rangle$ is an interval, i and k are non-negative integers, and j is an integer or ω, such that $i \leq k \preceq j \leq |\sigma|$. We use (σ, i, k, j) to mean that a term or formula is interpreted over a subinterval $\sigma_{(i..j)}$ 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 as $\mathcal{I}[e]$, is a value in $\mathcal{D}$ or *nil*. It is defined by structural induction, shown in Fig .1.

$$\begin{aligned}
\mathcal{I}[x] &= s_k[x] = I_v^k[x] \\
\mathcal{I}[\bigcirc e] &= \begin{cases} (\sigma, i, k+1, j)[e] & \text{if } k < j \\ \mathit{nil} & \text{otherwise} \end{cases} \\
\mathcal{I}[\ominus e] &= \begin{cases} (\sigma, i, k-1, j)[e] & \text{if } i < k \\ \mathit{nil} & \text{otherwise} \end{cases} \\
\mathcal{I}[f(e_1, \ldots, e_m)] &= \begin{cases} \mathcal{I}[f](\mathcal{I}[e_1], \ldots, \mathcal{I}[e_m]) & \text{if } \mathcal{I}[e_h] \neq \mathit{nil} \text{ for all } h \\ \mathit{nil} & \text{otherwise} \end{cases}
\end{aligned}$$

Fig. 1. Interpretation of PTL terms

The satisfaction relation for formulas $\models$ is inductively defined as follows.

1. $\mathcal{I} \models r$ if $s_k[r] = I_p^k[r] = \mathsf{true}$.
2. $\mathcal{I} \models e_1 = e_2$ if $\mathcal{I}[e_1] = \mathcal{I}[e_2]$.

3. $\mathcal{I} \models P(e_1, \ldots, e_m)$ if $\mathcal{I}[e_h] \neq nil$ for $1 \leq h \leq m$ and $\mathcal{I}[P]\,(\mathcal{I}[e_1], \ldots, \mathcal{I}[e_m]) = \mathsf{true}$.
4. $\mathcal{I} \models \neg p$ if $\mathcal{I} \not\models p$.
5. $\mathcal{I} \models p_1 \wedge p_2$ if $\mathcal{I} \models p_1$ and $\mathcal{I} \models p_2$.
6. $\mathcal{I} \models \exists v : p$ if for some interval σ' which has the same length as σ, $(\sigma', i, k, j) \models p$ and the only difference between σ and σ' can be the values of the variable v at the state k.
7. $\mathcal{I} \models \bigcirc p$ if $k < j$ and $(\sigma, i, k+1, j) \models p$.
8. $\mathcal{I} \models (p_1, \ldots, p_m)\ \mathsf{prj}\ q$ if there exist integers $k = k_0 \leq k_1 \leq \ldots \leq k_m \leq j$ such that $(\sigma, i, k_0, k_1) \models p_1$, $(\sigma, k_{l-1}, k_{l-1}, k_l) \models p_l$ (for $1 < l \leq m$), and $(\sigma', 0, 0, |\sigma'|) \models q$ for one of the following σ':
   (a) $k_m < j$ and $\sigma' = \sigma \downarrow (k_0, \ldots, k_m) \bullet \sigma_{(k_m+1..j)}$
   (b) $k_m = j$ and $\sigma' = \sigma \downarrow (k_0, \ldots, k_h)$ for some $0 \leq h \leq m$.

A formula p is said to be:

1. *satisfied* by an interval σ, denoted as $\sigma \models p$, if $(\sigma, 0, 0, |\sigma|) \models p$.
2. *satisfiable* if $\sigma \models p$ for some σ.
3. *valid*, denoted as $\models p$, if $\sigma \models p$ for all σ.
4. *equivalent* to another formula q, denoted as $p \equiv q$, if $\models \Box(p \leftrightarrow q)$.

The connectors $\vee$, $\rightarrow$ and $\leftrightarrow$ are defined as usual. In particular, the abbreviations $\mathsf{true} \stackrel{\text{def}}{=} p \vee \neg p$ and $\mathsf{false} \stackrel{\text{def}}{=} p \wedge \neg p$ for any formula p. In Fig. 2 derived formulas and composite predicates are shown.

$$
\begin{array}{lcl}
\mathsf{empty} & \stackrel{\text{def}}{=} & \neg \bigcirc \mathsf{true} \\
p; q & \stackrel{\text{def}}{=} & (p, q)\ \mathsf{prj}\ \mathsf{empty} \\
\Box p & \stackrel{\text{def}}{=} & \neg \Diamond \neg p \\
\mathsf{keep}(p) & \stackrel{\text{def}}{=} & \Box(\neg \mathsf{empty} \rightarrow p) \\
\mathsf{skip} & \stackrel{\text{def}}{=} & \bigcirc \mathsf{empty} \\
\mathsf{len}(0) & \stackrel{\text{def}}{=} & \mathsf{empty} \\
p^* & \stackrel{\text{def}}{=} & \mathsf{empty} \vee (p; p^*) \vee p \wedge \Box \mathsf{more}
\end{array}
\qquad
\begin{array}{lcl}
\mathsf{more} & \stackrel{\text{def}}{=} & \neg \mathsf{empty} \\
\Diamond p & \stackrel{\text{def}}{=} & \mathsf{true}; p \\
\mathsf{halt}(p) & \stackrel{\text{def}}{=} & \Box(\mathsf{empty} \leftrightarrow p) \\
\mathsf{fin}(p) & \stackrel{\text{def}}{=} & \Box(\mathsf{empty} \rightarrow p) \\
\mathsf{len}(n) & \stackrel{\text{def}}{=} & \bigcirc \mathsf{len}(n-1) \quad \text{for } n > 0
\end{array}
$$

Fig. 2. Derived formulas and composite predicates

In order to avoid an excessive number of parentheses, the following precedence rules are used as shown in Table 1. An operator with a smaller number has a higher precedence, while operators with the same number have the same precedence.

Table 1. Precedence rules of PTL

1		$\neg$		2	$\bigcirc$	$\ominus$	$\Diamond$	$\Box$	3	$\wedge$	4	$\vee$
5	:=	=		6	$\rightarrow$	$\leftrightarrow$			7	prj	8	;

2.2 MSVL

Let n range over integers and x range over variables. MSVL arithmetic expressions e and boolean expressions b are PTL terms and formulas, respectively [5]. They are given by the following grammar:

$$e ::= n \mid x \mid \bigcirc x \mid \ominus x \mid e_1 + e_2 \mid e_1 - e_2 \mid e_1 * e_2 \mid e_1 / e_2 \mid e_1 \; mod \; e_2$$
$$b ::= \mathsf{true} \mid \mathsf{false} \mid e_1 = e_2 \mid e_1 > e_2 \mid e_1 \geq e_2 \mid e_1 < e_2 \mid e_1 \leq e_2 \mid \neg b \mid b_1 \wedge b_2$$

The elementary statements p, q of MSVL are PTL formulas and defined as follows.

Assignment:	$x = e$
P-I-Assignment:	$x \Leftarrow e \stackrel{\text{def}}{=} x = e \wedge r_x$
Unit Assignment:	$x := e \stackrel{\text{def}}{=} \mathsf{skip} \wedge \bigcirc x \Leftarrow e$
Sequential Composition:	$p; q$
Conditional Choice:	$\texttt{if}\ b\ \texttt{then}\ p\ \texttt{else}\ q \stackrel{\text{def}}{=} (b \rightarrow p) \wedge (\neg b \rightarrow q)$
While Loop:	$\texttt{while}\ b\ \texttt{do}\ p \stackrel{\text{def}}{=} (b \wedge p)^* \wedge \Box(\mathsf{empty} \rightarrow \neg b)$
Conjunction:	$p \wedge q$
Selection:	$p \vee q$
Parallel Composition:	$p \parallel q \stackrel{\text{def}}{=} p \wedge (q; \mathsf{true}) \vee q \wedge (p; \mathsf{true})$
Next:	$\bigcirc p$
Always:	$\Box p$
Termination:	empty
Local variable:	$\exists x : p$
State Frame:	$\mathsf{lbf}(x) \stackrel{\text{def}}{=} \neg r_x \rightarrow \exists z : (\ominus x = z \wedge x = z)$
Interval Frame:	$\mathsf{frame}(x) \stackrel{\text{def}}{=} \Box(\mathsf{more} \rightarrow \bigcirc \mathsf{lbf}(x))$
Projection:	$(p_1, \ldots, p_m)\ \mathsf{prj}\ q$
Await:	$\mathsf{await}(b) \stackrel{\text{def}}{=} (\mathsf{frame}(x_1) \wedge \ldots \wedge \mathsf{frame}(x_h)) \wedge \Box(\mathsf{empty} \leftrightarrow b)$, where $x_1, \ldots, x_h$ are the variables that occur in b

Among the statements, $x = e$, $x := e$, $x \Leftarrow e$, empty, $\mathsf{lbf}(x)$, and $\mathsf{frame}(x)$ are basic statements, while the others are composite statements. An assignment $x = e$ means that the value of x equals the value of e, while a unit assignment $x := e$ specifies the value of x by e and the length of the interval by 1. A positive immediate assignment (P-I-Assignment) $x \Leftarrow e$ indicates that the value of x equals the value of e and that the assignment flag r_x for x is true. A sequential composition $p; q$ indicates that p is executed from the current state until its termination when q is executed from. Statements of conditional choice $\texttt{if}\ b\ \texttt{then}\ p\ \texttt{else}\ q$ and while loop $\texttt{while}\ b\ \texttt{do}\ p$ are the same as they are in conventional imperative languages. A conjunction $p \wedge q$ means that p and q are executed concurrently and share all the variables during the mutual execution, while a selection $p \vee q$ means either p or q is executed. Different from a conjunction, a parallel composition allows both processes to specify their own intervals, e.g. $\mathsf{len}(3) \parallel \mathsf{len}(4)$ can be satisfied but $\mathsf{len}(3) \wedge \mathsf{len}(4)$ is always false. A next statement $\bigcirc p$ means that p holds at the next state, while an always statement

$\Box p$ means that p holds at all states over the current interval. The termination statement empty means that the current state is the final state of the interval. An existential quantification $\exists x : p$ intends to hide x within p. A state frame $\mathsf{lbf}(x)$ means the value of x in the current state equals the value of x in the previous state if no assignment to x is encountered, while $\mathsf{frame}(x)$ indicates that the value of variable x always keeps its old value over an interval if no assignment to x is encountered. A projection statement can be thought of as a special parallel execution that is performed on different time scales. Specifically, $(p_1, \ldots, p_m)\ \mathsf{prj}\ q$ means that q is executed in parallel with $p_1, \ldots, p_m$ over an interval obtained by taking the endpoints of the intervals over which the p_i's are executed. In particular, the sequence of p_i's and q may terminate at different time points. Finally, an await statement $\mathsf{await}(b)$ simply waits until b becomes true, without changing any variables.

The precedence rules of MSVL statements are listed in Table 2, where 1 means highest and 12 means lowest.

Table 2. Precedence rules of MSVL

1		$\neg$	2	$\bigcirc$	$\ominus$	$\Box$	3	$*$	/	*mod*	4	$+$	$-$	5	$>$	$\geq$	$<$	$\leq$	6	$\exists$
7	$=$	$\Leftarrow$	$:=$	8	$\wedge$		9	$\vee$	$\|\|$		10	$\rightarrow$	$\leftrightarrow$	11	prj				12	;

3 Typed MSVL

3.1 Data Domain

In Sect. 2, for a state $s = (I_v, I_p)$ and a variable x defined in s, $I_v[x] \in \mathcal{D}$. If a variable y is irrelevant to (i.e. undefined in) s, we write $I_v[y] = nil$. So, for any interpretation $\mathcal{I}$ and variable x, $\mathcal{I}[x] \in \mathcal{D} \cup \{nil\}$. In order to extend the interpretation I_v of variables to typed values, we need to enlarge the data domain D. We introduce into MSVL a set $\mathcal{T}$ of types, including

- basic types: int, float, char,
- list types: int⟨⟩, float⟨⟩, char⟨⟩,
- array types: int[], float[], char[].

The set of values of each basic type are defined as follows.

- int: $\mathbb{Z}$
- float: $\mathtt{F} \stackrel{\text{def}}{=} \{n.d_1d_2\cdots d_m \mid m \in \mathbb{N}, n \in \mathbb{Z}, d_i \in \{0,\ldots,9\} for 1 \leq i \leq m\}$
- char: $\mathtt{C} \stackrel{\text{def}}{=} \{'a', \ldots, 'z', 'A', \ldots, 'Z', '0', \ldots, '9', ' ', '!', '@', '\sharp', '\$', \ldots\}$

Consider typed values (v, T), i.e. values labeled by their types, where $T \in \mathcal{T}$. We define the set of typed values for each type. For a set S, S^n denotes the set of lists of length n on S ($n \in \mathbb{N}$), and S^* denotes the set of lists on S.

- $\text{Int} \stackrel{\text{def}}{=} \mathbb{Z} \times \{\text{int}\}$, $\text{Int}\langle\rangle \stackrel{\text{def}}{=} \mathbb{Z}^* \times \{\text{int}\langle\rangle\}$,
  $\text{Int}[n] \stackrel{\text{def}}{=} \mathbb{Z}^n \times \{\text{int}[\,]\}$ for $n \geq 1$, $\text{Int}[\,] \stackrel{\text{def}}{=} \bigcup_{n\geq 1} \text{Int}[n]$
- $\text{Float} \stackrel{\text{def}}{=} \mathtt{F} \times \{\text{float}\}$, $\text{Float}\langle\rangle \stackrel{\text{def}}{=} \mathtt{F}^* \times \{\text{float}\langle\rangle\}$,
  $\text{Float}[n] \stackrel{\text{def}}{=} \mathtt{F}^n \times \{\text{float}[\,]\}$ for $n \geq 1$, $\text{Float}[\,] \stackrel{\text{def}}{=} \bigcup_{n\geq 1} \text{Float}[n]$
- $\text{Char} \stackrel{\text{def}}{=} \mathtt{C} \times \{\text{char}\}$, $\text{Char}\langle\rangle \stackrel{\text{def}}{=} \mathtt{C}^* \times \{\text{char}\langle\rangle\}$,
  $\text{Char}[n] \stackrel{\text{def}}{=} \mathtt{C}^n \times \{\text{char}[\,]\}$ for $n \geq 1$, $\text{Char}[\,] \stackrel{\text{def}}{=} \bigcup_{n\geq 1} \text{Char}[n]$

The data domain D of variables is the union of these sets.

$$\mathcal{D} \stackrel{\text{def}}{=} \text{Int} \cup \text{Int}\langle\rangle \cup \text{Int}[\,] \cup \text{Float} \cup \text{Float}\langle\rangle \cup \text{Float}[\,] \cup \text{Char} \cup \text{Char}\langle\rangle \cup \text{Char}[\,]$$

3.2 Typed Functions and Predicates

Each function and predicate has not only a fixed arity but also a fixed typed. Specifically, a function f of arity m has a type $T_1 \times \ldots \times T_m \to T$, and a predicate of arity n has a type $T_1 \times \ldots \times T_n \to \mathbb{B}$, where each T_i and T is a type in $\mathcal{T}$. For example, the original MSVL function $\cdot + \cdot$ and predicate $\cdot > \cdot$ are applied to integers, and their types are denoted as $\cdot + \cdot : \text{int} \times \text{int} \to \text{int}$ and $\cdot > \cdot : \text{int} \times \text{int} \to \mathbb{B}$, respectively.

Since we extend the interpretation I_v of variables to typed values, we also need to extend the application of functions and predicates to typed values. The extension is straightforward. For a function $f : T_1 \times \ldots \times T_m \to T$ with $f(v_1, \ldots, v_m) = v$, we now have $f((v_1, T_1), \ldots, (v_m, T_m)) = (v, T)$, and for a predicate $P : T_1 \times \ldots \times T_m \to \mathbb{B}$ with $P(v_1, \ldots, v_m) = \mathsf{true}$ (or false), we now have $P((v_1, T_1), \ldots, (v_m, T_m)) = \mathsf{true}$ (or false). Implicitly, a function with ill-typed parameters evaluates to *nil*, and a predicate with ill-typed parameters is interpreted as false. As a result, $\mathcal{I}[e] \in \mathcal{D} \cup \{nil\}$ for any expression e and interpretation $\mathcal{I}$, according to the evaluation rules defined in Fig. 1. Notice that even for type-correct and defined parameters, the result of a function can be undefined. For example, $(3, \text{int})/(0, \text{int}) = nil$, $hd((\langle\rangle, \text{int}\langle\rangle)) = nil$. However, this will not cause any problem.

A constant c is regarded as a 0-arity function. The kinds of constants allowed in MSVL programs are listed below, together with their interpretations.

- Integers, float numbers and characters, e.g. $\mathcal{I}[8] = (8, \text{int})$, $\mathcal{I}[3.1] = (3.1, \text{float})$ and $\mathcal{I}['a'] = ('a', \text{char})$.
- Non-empty lists, e.g. $\mathcal{I}[\langle 'x', 'y'\rangle] = (\langle 'x', 'y'\rangle, \text{char}\langle\rangle)$.
- Empty lists, $\mathcal{I}[\langle\rangle_i] = (\langle\rangle, \text{int}\langle\rangle)$, $\mathcal{I}[\langle\rangle_f] = (\langle\rangle, \text{float}\langle\rangle)$ and $\mathcal{I}[\langle\rangle_c] = (\langle\rangle, \text{char}\langle\rangle)$.

Notice that we discriminate empty lists of integers ($\langle\rangle_i$), float numbers ($\langle\rangle_f$) and characters ($\langle\rangle_c$). This is to ensure that every expression has a fixed type.

The original functions $+, -, *, /, mod$ and predicates $>, \geq, <, \leq$ are all for the integer type. We need to define operations for new types. First, we define $+$ (and then $-$, $*$ and $/$) of float numbers. One way is to define a specific function

$$\cdot +_f \cdot : \text{float} \times \text{float} \to \text{float}$$

for the float type. A more convenient approach is to unify the two addition operations into one.

$$\cdot + \cdot : (\text{int} \times \text{int} \to \text{int}) \cup (\text{float} \times \text{float} \to \text{float})$$

That is, we allow a function f (or predicate P) to have a union of more than one fixed types. Each application of f (or P) takes one of these types.

Arithmetic operators. Besides $+$, we do the same extension on $-$, $*$ and $/$ so that

$$\cdot + \cdot, \cdot - \cdot, \cdot * \cdot, \cdot / \cdot : (\text{int} \times \text{int} \to \text{int}) \cup (\text{float} \times \text{float} \to \text{float})$$

We keep the function *mod* : int × int → int. We also extend the predicates $>, \geq, <, \leq$ to both integer of float types.

$$\cdot > \cdot, \cdot \geq \cdot, \cdot < \cdot, \cdot \leq \cdot : (\text{int} \times \text{int} \to \mathbb{B}) \cup (\text{float} \times \text{float} \to \mathbb{B})$$

Type cast. We define two functions for type cast between float numbers and integers.

$$\begin{array}{ll} (\text{int})\cdot : \text{float} \to \text{int} & (n.d_1 d_2 \cdots d_m, \text{float}) \mapsto (n, \text{int}) \\ (\text{float})\cdot : \text{int} \to \text{float} & (n, \text{int}) \mapsto (n., \text{float}) \end{array}$$

Array and list operations. We define a set of standard operations for arrays and lists. For an array a, the operation $a[i]$ returns its ith element.

$$\begin{array}{l} \cdot[\cdot] : (\text{int}[\,] \times \text{int} \to \text{int}) \cup (\text{float}[\,] \times \text{int} \to \text{float}) \cup (\text{char}[\,] \times \text{int} \to \text{char}) \\ \quad (\langle c_0, \ldots, c_k \rangle, T[\,]), (i, \text{int}) \mapsto (c_i, T) \qquad i \in \{0, \ldots, k\} \\ \quad (\langle c_0, \ldots, c_k \rangle, T[\,]), (i, \text{int}) \mapsto \mathit{nil} \qquad i \notin \{0, \ldots, k\} \\ \quad k \in \mathbb{N}, T \in \{\text{int}, \text{float}, \text{char}\} \end{array}$$

For a list l, the operation $|l|$ returns the length of l.

$$\begin{array}{l} |\cdot| : (\text{int}\langle\rangle \to \text{int}) \cup (\text{float}\langle\rangle \to \text{int}) \cup (\text{char}\langle\rangle \to \text{int}) \\ \quad (\langle c_1, \ldots, c_k \rangle, T) \mapsto (k, \text{int}) \qquad k \in \mathbb{N}, T \in \{\text{int}\langle\rangle, \text{float}\langle\rangle, \text{char}\langle\rangle\} \end{array}$$

Besides, the operations $hd(l)$ and $tl(l)$ return the head and tail of l, respectively.

$$\begin{array}{ll} \mathit{hd} : & (\text{int}\langle\rangle \to \text{int}) \cup (\text{float}\langle\rangle \to \text{float}) \cup (\text{char}\langle\rangle \to \text{char}) \\ & (\langle\rangle, T\langle\rangle) \mapsto \mathit{nil} \\ & (\langle c_0, c_1, \ldots, c_k \rangle, T\langle\rangle) \mapsto (c_0, T) \qquad k \in \mathbb{N}, T \in \{\text{int}, \text{float}, \text{char}\} \\ \mathit{tl} : & (\text{int}\langle\rangle \to \text{int}\langle\rangle) \cup (\text{float}\langle\rangle \to \text{float}\langle\rangle) \cup (\text{char}\langle\rangle \to \text{char}\langle\rangle) \\ & (\langle\rangle, T) \mapsto \mathit{nil} \\ & (\langle c_0, c_1, \ldots, c_k \rangle, T) \mapsto (\langle c_1, \ldots, c_k \rangle, T) \quad k \in \mathbb{N}, T \in \{\text{int}\langle\rangle, \text{float}\langle\rangle, \text{char}\langle\rangle\} \end{array}$$

For two lists l_1 and l_2 of the same type, the operations $l_1 \bullet l_2$ and $l_1 \circ l_2$ calculates the concatenation and fusion of l_1 and l_2, respectively. They are defined as follows.

$$\begin{array}{l} \cdot \bullet \cdot : (\text{int}\langle\rangle \times \text{int}\langle\rangle \to \text{int}\langle\rangle) \cup (\text{float}\langle\rangle \times \text{float}\langle\rangle \to \text{float}\langle\rangle) \\ \quad \cup (\text{char}\langle\rangle \times \text{char}\langle\rangle \to \text{char}\langle\rangle) \\ \quad (\langle c_1, \ldots, c_j \rangle, T), (\langle d_1, \ldots, d_k \rangle, T) \mapsto (\langle c_1, \ldots, c_j, d_1, \ldots, d_k \rangle, T) \\ \quad j, k \in \mathbb{N}, T \in \{\text{int}\langle\rangle, \text{float}\langle\rangle, \text{char}\langle\rangle\} \end{array}$$

$$
\begin{aligned}
&\cdot \circ \cdot : (\mathrm{int}\langle\rangle \times \mathrm{int}\langle\rangle \to \mathrm{int}\langle\rangle) \cup (\mathrm{float}\langle\rangle \times \mathrm{float}\langle\rangle \to \mathrm{float}\langle\rangle) \\
&\quad \cup (\mathrm{char}\langle\rangle \times \mathrm{char}\langle\rangle \to \mathrm{char}\langle\rangle) \\
&\quad (\langle\rangle, T), (\langle d_1, \ldots, d_k\rangle, T) \mapsto (\langle d_1, \ldots, d_k\rangle, T) \\
&\quad (\langle c_1, \ldots, c_j, c\rangle, T), (\langle\rangle, T) \mapsto (\langle c_1, \ldots, c_j, c\rangle, T) \\
&\quad (\langle c_1, \ldots, c_j, c\rangle, T), (\langle c, d_1, \ldots, d_k\rangle, T) \mapsto (\langle c_1, \ldots, c_j, c, d_1, \ldots, d_k\rangle, T) \\
&\quad (\langle c_1, \ldots, c_j, c\rangle, T), (\langle d, d_1, \ldots, d_k\rangle, T) \mapsto \mathit{nil} \qquad d \neq c \\
&\quad j, k \in \mathbb{N}, T \in \{\mathrm{int}\langle\rangle, \mathrm{float}\langle\rangle, \mathrm{char}\langle\rangle\}
\end{aligned}
$$

Other. Besides the above operations, we define an auxiliary predicate Def(). For an expression e, Def(e) means e is defined, i.e., the value of e is not *nil*.

$$
\begin{aligned}
&\mathrm{Def} : (\mathrm{int} \to \mathbb{B}) \cup (\mathrm{float} \to \mathbb{B}) \cup (\mathrm{char} \to \mathbb{B}) \cup (\mathrm{int}\langle\rangle \to \mathbb{B}) \cup (\mathrm{float}\langle\rangle \to \mathbb{B}) \\
&\quad \cup (\mathrm{char}\langle\rangle \to \mathbb{B}) \\
&\quad (c, T) \mapsto \mathsf{true} \qquad T \in \{\mathrm{int}, \mathrm{float}, \mathrm{char}, \mathrm{int}\langle\rangle, \mathrm{float}\langle\rangle, \mathrm{char}\langle\rangle\}
\end{aligned}
$$

Because an expression can evaluate to *nil*, the meaning of some predicates or formulas may not precisely reflect our intuition. For example, $e_1 \leq e_2 \equiv \neg(e_1 > e_2)$ is not valid in MSVL. Instead, $e_1 \leq e_2 \equiv \mathrm{Def}(e_1) \wedge \mathrm{Def}(e_2) \wedge \neg(e_1 > e_2)$. This is why the predicate Def() is useful. With the predicate, another equivalent characterization of $e_1 \leq e_2$ is $\mathrm{Def}(e_1) \wedge \mathrm{Def}(e_2) \wedge (e_1 < e_2 \vee e_1 = e_2)$.

3.3 Type Declaration Statement

Notice that when declaring an array we need to give the specific number of elements of the array, e.g. int[5] a. So the set of types $\mathcal{T}_d$ that are used in type declarations is slightly different from the set $\mathcal{T}$:

$$
\begin{aligned}
\mathcal{T}_d \ \stackrel{\mathrm{def}}{=}\ &\{\mathrm{int}, \mathrm{float}, \mathrm{char}, \mathrm{int}\langle\rangle, \mathrm{float}\langle\rangle, \mathrm{char}\langle\rangle, \\
&\mathrm{int}[1], \mathrm{int}[2], \ldots, \mathrm{float}[1], \mathrm{float}[2], \ldots, \mathrm{char}[1], \mathrm{char}[2], \ldots\}.
\end{aligned}
$$

We define predicates $\mathrm{is}_T(\cdot)$, which means "is of type T", for each type $T \in \mathcal{T}_d$.

1. For each basic type $T \in \{\mathrm{int}, \mathrm{float}, \mathrm{char}\}$, $\mathrm{is}_T : T \to \mathbb{B}$ with $(c, T) \mapsto \mathsf{true}$,
2. for each list type $T\langle\rangle \in \{\mathrm{int}\langle\rangle, \mathrm{float}\langle\rangle, \mathrm{char}\langle\rangle\}$, $\mathrm{is}_{T\langle\rangle} : T\langle\rangle \to \mathbb{B}$ with $(c, T\langle\rangle) \mapsto \mathsf{true}$, and
3. for each array type $T[n] \in \{\mathrm{int}[1], \mathrm{int}[2], \ldots, \mathrm{float}[1], \ldots, \mathrm{char}[1], \ldots\}$, $\mathrm{is}_{T[n]} : T[\,] \to \mathbb{B}$ with $(c, T[\,]) \mapsto \mathsf{true}$ iff $|c| = n$.

Using these predicates, we define the type declaration statement as a derived PTL formula.

$$T\,x \ \stackrel{\mathrm{def}}{=}\ \Box \mathrm{is}_T(x)$$

4 Implementation Mechanisms

This section focuses on the implementation mechanisms for the type declaration statement which plays an important role in program execution in the MSVL interpreter. In order to carry out the formal verification and analysis of programs

in a rigorous way, an operational semantics for the type declaration statement are needed. For reducing (executing) MSVL programs, we divide the reduction process into two phases [14]: one for state reduction and the other for interval reduction. The state reduction is mainly on how to transform a program into its normal form [5]. The interval reduction is concerned with a program from one state to another. The state reduction has to change the normal form of programs after the introduction of the type declaration statement, while the interval reduction remains unchanged.

4.1 Normal Form of Programs

Definition 1. *A typed MSVL program q is in normal form if*

$$q \overset{\text{def}}{=} (\bigvee_{i=1}^{k} q_{ei} \wedge \mathsf{empty}) \vee (\bigvee_{j=1}^{h} q_{cj} \wedge \bigcirc q_{fj})$$

where $k + h \geq 1$ and the following hold:

1. *each q_{ei} and q_{cj} is either true or a state formula of the form $p_1 \wedge \ldots \wedge p_m$ $(m \geq 1)$ such that each p_l $(1 \leq l \leq m)$ is either $\mathrm{is}_T(x)$ with $x \in \mathcal{V}$, $T \in \mathcal{T}_d$, or $x = e$ with $e \in D$, or r_x, or $\neg r_x$.*
2. *q_{fj} is an internal program, that is, one in which variables may refer to the previous states but not beyond the first state of the current interval over which the program is executed.*

When a typed MSVL program q is deterministic $k + h = 1$ holds, otherwise $k + h > 1$ does. We call conjuncts, $\bigvee_{i=1}^{k} q_{ei} \wedge \mathit{empty}$, $\bigvee_{j=1}^{h} q_{cj} \wedge \bigcirc q_{fj}$ basic products: the former is called terminal products whereas the latter is called future products. Further we call q_{ei} and q_{cj} present components which are executed at the current state, and $\bigcirc q_{fj}$ future components executed in the subsequent states. An important conclusion is that any typed MSVL program including type declaration statements can be reduced to its normal form. Therefore, execute programs in MSVL is to transform them logically equivalent to their normal forms.

Let p be an MSVL program augmented with type declarations. There is a program q in normal form such that $p \equiv q$.

The proof proceeds by induction on the structure of statements. The proof of MSVL statements without type declarations can be found in [12,13].

The proof of the type declaration statement $T\,x$ is given as follows.

$$\begin{aligned}
T\,x &\equiv \Box \mathrm{is}_T(x) \\
&\equiv \Box \mathrm{is}_T(x) \wedge (\mathsf{empty} \vee \neg \mathsf{empty}) && (i)\\
&\equiv \Box \mathrm{is}_T(x) \wedge \mathsf{empty} \vee \Box \mathrm{is}_T(x) \wedge \neg \mathsf{empty} && (ii)\\
&\equiv \mathrm{is}_T(x) \wedge \mathsf{empty} \vee \Box \mathrm{is}_T(x) \wedge \neg \mathsf{empty} && (iii)\\
&\equiv \mathrm{is}_T(x) \wedge \mathsf{empty} \vee \Box \mathrm{is}_T(x) \wedge \mathsf{more} && (iv)\\
&\equiv \mathrm{is}_T(x) \wedge \mathsf{empty} \vee \mathrm{is}_T(x) \wedge \bigcirc \Box \mathrm{is}_T(x) && (v)
\end{aligned}$$

In the above: (i) follows from T1 in [14]; (ii) from Theorem 2.1 in [13]; (iii) from Law7 in [14]; (iv) from the definition of more; and (v) from Law8 in [14].

4.2 MSVL Interpreter

Microsoft Visual C++ has been used to implement an MSVL interpreter. The flow chart of the interpreter is shown in Fig. 3. The lexical analyzer and parser are implemented with flex and bison. In the state reduction based on the normal form an MSVL program can be rewritten by the reducer module to a logically equivalent formula $Present \wedge Remains$. The formula $Present$ is executed at the current state. It consists of true, false, empty, immediate variable assignments or variable input/output. In the interval reduction the formula $Remains$ is executed in the succeeding state if it exists. The program editor, data input, output view modules are used to deal with input and output. An MSVL program is inputted into the interpreter and executed in a sequence of states to try to find its model. If the program is transformed to true at the final state, its model is found and it is satisfiable, otherwise it has no model and is unsatisfiable.

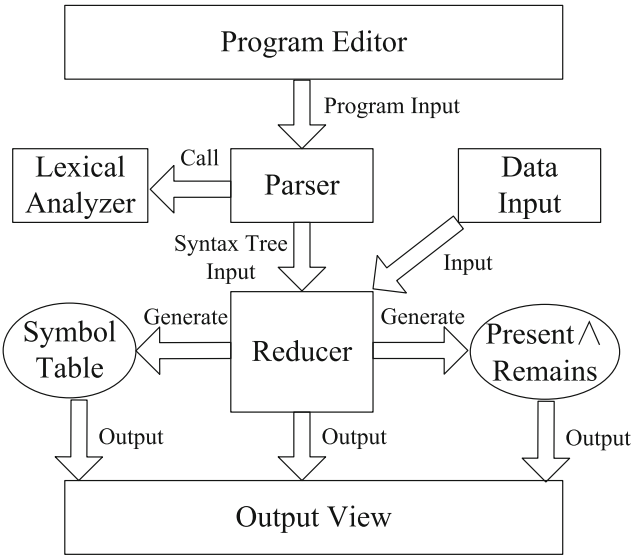
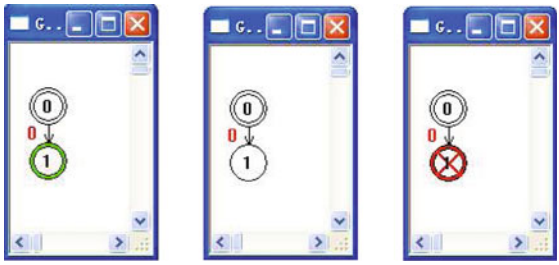


Fig. 3. Interpreter structure

The MSVL interpreter is able to work in an modeling, simulation or verification mode. In the first mode, an MSVL program is used to describe a system and executed in the interpreter. All the models of the system are presented as an Normal Form Graph (NFG) [12]. As show in Fig. 4(a) a path in the NFG ends with a bicyclic node is a model of the system. The simulation mode is a little different with the modeling mode, and the interpreter outputs only one



(a) (b) (c)

Fig. 4. Three types of nodes. (a)modeling: a path. (b)verification: a satisfiable path. (c)verification: an unsatisfiable path.

path in the NFG according to the MSVL's minimal model semantics [12]. The interpreter can also work in the verification mode. Given an MSVL program to describe a system, and a PPTL formula to describe its property, the interpreter can automatically verify whether or not the system satisfies the property. If the system is unsatisfied with the property, the interpreter will point out a counterexample. As shown in Fig. 4(b) a satisfiable path in the NFG ends with a circular node, while as shown in Fig. 4(c) an unsatisfiable path in the NFG ends with a terminative node. It is worth pointing out that the formalization of types only extends the data domain to typed values. It does not change the structures of MSVL programs or the finiteness of program states. Therefore, we can still translate a model checking problem into a satisfiability problem in PPTL since finite-state MSVL programs are equivalent to PPTL formulas [5].

5 An Application

In Fig. 5, two integer linked list both include three nodes. The integers in the first list from the head to the tail are 10, 20 and 30, and the integers in the second one are in the opposite direction. The following MSVL program executes an in-place reversal of the first integer list and gets the second one. The pointer operations & and * are defined in [15].

```
frame(node1, node2, node3, p, q, r, head, tail) and (
    int[2] node1, node2, node3;
    pointer p,q,r;
    node1[0] = 10 and node1[1] := -1;
    node2[0] = node1[0] + 10 and node2[1] := -1;
    node3[0] = (int)30.0 and node3[1] := -1;
    node2[1] := &node3;     node1[1] := &node2;
    q :=& node1;    head := *q[0];
    while(q != -1){ tail := *q[0];      q := *q[1]  };
    p :=& node1;    q := -1;
```

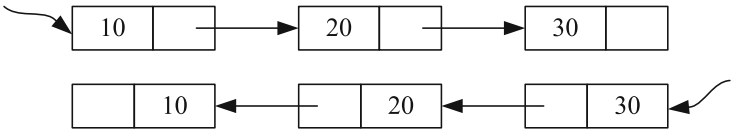


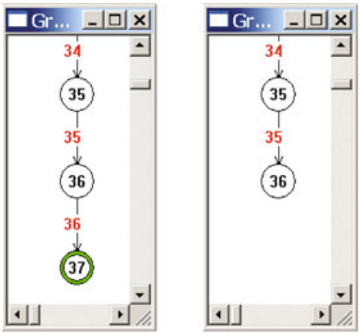
Fig. 5. In-place reversal of an integer list

```
    while(p != -1){ r := *p[1]; *p[1] := q; q := p; p := r   };
    head := *q[0];
    while(q != -1){ tail := *q[0];      q := *q[1]  }
)
```

As showed in Fig. 6(a), the program executes successfully in the modeling mode and outputs 37 states. Before verifying the program, its properties have been formalized as follows. The proposition `prop1` is `head = 10` indicates the list head is 10, and `prop2` is `tail = 30` indicates the list tail is 30. The meanings of `prop3` and `prop4` are similar. The desirable property of the former program is described by a PPTL formula $\Diamond(prop1 \wedge prop2) \wedge \Box(\mathsf{empty} \rightarrow prop3 \wedge prop4)$. The property is coded as follows.

```
</
    define prop1: head = 10;    define prop2: tail = 30;
    define prop3: head = 30;    define prop4: tail = 10;
    som(prop1 and prop2) and always(empty -> prop3 and prop4)
/>
```

The MSVL interpreter executes the program with properties in the verification mode and the results are showed in Fig. 6(b). The final node of the execution path is not a bicyclic node, and it shows that the desirable property is satisfied.



(a) (b)

Fig. 6. Program execution. (a) modeling result. (b) verification result.

6 Conclusions

In this paper, we provide a formalization and implementation of types in the temporal logic programming language MSVL. The data domain of MSVL is enlarged to include typed values. Typed functions and predicates, and the type declaration statement are defined. The MSVL interpreter implementation mechanisms based on the notion of normal form are also given. In the near future, much research work is to be done to investigate the operational and axiomatic semantics of types in MSVL. In addition, we will also try to model and verify some larger examples within our approach.

References

1. Allen Emerson, E.: Temporal and modal logic. In: Handbook of Theoretical Computer Science, pp. 995–1072. Elsevier, Amsterdam (1995)
2. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
3. Baier, C., Katoen, J.-P.: Principles of Model Checking. MIT Press, Cambridge (2008)
4. Duan, Z., Maciej, K.: A framed temporal logic programming language. J. Comput. Sci. Technol. **19**, 341–351 (2004)
5. Duan, Z., Tian, C.: A unified model checking approach with projection temporal logic. In: Liu, S., Maibaum, T., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 167–186. Springer, Heidelberg (2008)
6. Tian, C., Duan, Z.: Expressiveness of propositional projection temporal logic with star. Theor. Comput. Sci. **412**, 1729–1744 (2011)
7. Cau, A., Moszkowski, B., Zedan, H.: Itl and tempura home page on the web. http://www.cse.dmu.ac.uk/STRL/ITL/ (2013)
8. Tang, Z.: Temporal Logic Program Designing and Engineering. Science Press, Beijing (1999)
9. Lamport, L.: The TLA Home Page. http://research.microsoft.com/en-us/um/people/lamport/tla/tla.html (2013)
10. Fisher, M.: MetateM: the story so far. In: Bordini, R.H., Dastani, M.M., Dix, J., El Fallah Seghrouchni, A. (eds.) PROMAS 2005. LNCS (LNAI), vol. 3862, pp. 3–22. Springer, Heidelberg (2006)
11. Zhou, S., Zedan, H., Cau, A.: Run-time analysis of time-critical systems. J. Syst. Archit. **51**, 331–345 (2005)
12. Duan, Z., Yang, X., Koutny, M.: Framed temporal logic programming. Sci. Comput. Program. **70**, 31–61 (2008)
13. Duan, Z.: Temporal Logic and Temporal Logic Programming. Science Press, Beijing (2006)
14. Yang, X., Duan, Z.: Operational semantics of framed tempura. J. Logic Algebraic Program. **78**, 22–51 (2008)
15. Duan, Z., Wang, X.: Implementing pointer in temporal logic programming languages. In: Proceedings of SBMF 2006, pp. 171–184 (2006)