Model Checking C Programs with MSVL*

Yan Yu[1], Zhenhua Duan[1,**], Cong Tian[1], and Mengfei Yang[2]

[1] ICTT and ISN Lab, Xidian University, Xi'an, 710071, P.R. China
[2] China Academy of Space Technology, Beijing, 100094, P.R. China

Abstract. This paper presents an approach for model checking C programs with MSVL. To do so, we translate C programs into MSVL (modeling simulation and verification language) programs, and specify the desired property by a propositional projection temporal logic (PPTL) formula; then we employ the unified model checking approach to check whether the MSVL program satisfies the PPTL formula. If so, the program is correct; otherwise, a counterexample can be found. The translation algorithm from C to MSVL programs is introduced in details. In addition, an example is given to illustrate how the approach works.

Keywords: Temporal Logic, MSVL, Model Checking, Verification, Translation.

1 Introduction

C is one of the most widely used programming languages for the development of software systems. How to guarantee the correctness and reliability of C programs is a grand challenge to computer scientists and software engineers. In the past four decades, a number of testing techniques and verification approaches have been proposed for testing and verifying C programs with success. In particular, model checking is an automatic approach for verification[1–3] of C programs. With this approach, to verify a C program, an abstract model which describes the behaviors of the program must be extracted from the C program. Further, the property[4–6] to be verified can be specified by a temporal logic[7, 9, 10] (TL) formula. Then, a model checker is employed to check whether or not the model satisfies the property. If the model cannot satisfy the property, a counterexample is provided. As we can see, using model checking for verifying C programs suffers from the process which extracts the model from C programs since this process is not straightforward. On the other hand, with temporal logic programming, such as MSVL (modeling simulation verification language), approach for model checking, the behaviors of a system is described by an MSVL program and the property to be verified is described by a propositional projection temporal logic (PPTL) formula. Then, the unified model checking algorithm can be employed to verify whether or not the MSVL program satisfies the PPTL formula. As a matter of fact, with this approach, to verify C programs we have to rewrite the C program into MSVL programs, this is another burden for programmers. So we are motivated to work out a general translation

* This research is supported by NSFC Grants (No. 61133001, 6091004, 61272117, 61272118, 61003078, and 61202038), 973 Program (No.2010CB328102), and ISN Lab Grant No. ISN1102001.

** Corresponding author.

S. Liu (Ed.): SOFL 2012, LNCS 7787, pp. 87–103, 2013.

program which can automatically transfer a C program into a MSVL program, so that the model checking process can be automatically and directly conducted based on C programs.

To do so, we first analyze syntax and semantics of the C programs by means of lex and yacc within Paser Generator, and store the information in terms of a specified storage structure; second, we present an algorithm to achieve the translation from C statements to MSVL statements one by one; finally, all of statements in MSVL corresponding to C statements are generated. As a result, whenever a C program is input as a parameter, an equivalent MSVL program is produced. Therefore, to verify a C program, we only need to translate the C program into MSVL program and then verify the MSVL program based on the existing verification techniques.

The contributions of this paper are two-fold: (1) We design and implement a translator in C++ which translates any C program into an equivalent MSVL program. (2) We give an example to show how the translator works and conduct a model checking process to verify the property of the C program automatically.

The rest of this paper is organized as follows. In the next section, the unified model checking approach with MSVL and PPTL is briefly presented. In Section 3, an algorithm is formalized to implement the translation from C programs to MSVL programs. An example is given in Section 4 to show how the translation algorithm works, and how we use the algorithm to achieve our goals of directly verifying a C program. Finally, conclusions are drawn in Section 5.

2 Preliminaries

2.1 Projection Temporal Logic

2.1.1 Syntax

Let Π be a countable set of propositions, and V a countable set of typed static and dynamic variables. $B = \{true, false\}$ represents the boolean domain and D denotes all the data we need including integers, strings, lists, etc. The terms e and formulas p are given by the following grammar:

$$e ::= v \mid \bigcirc e \mid \ominus e \mid f(e_1, \ldots, e_n)$$
$$p ::= \pi \mid e_1 = e_2 \mid P(e_1, \ldots, e_n) \mid \neg p \mid p_1 \wedge p_2 \mid \exists v : p \mid \bigcirc p \mid (p_1, \ldots, p_m)\ \mathit{prj}\ p$$

where $\pi \in \Pi$ is a proposition, and v a dynamic or static variable. In $f(e_1, \ldots, e_n)$ and $P(e_1, \ldots, e_n)$, f is a function and P a predicate. It is assumed that the types of the terms are compatible with those of the arguments of f and P. A formula (term) is called a state formula (term) if it does not contain any temporal operators ($i.e. \bigcirc, \ominus$ and prj), otherwise it is a temporal formula (term).

2.1.2 Semantics

A state s is a pair of assignments (I_v, I_p) where for each variable $v \in V$ defines $s[v] = I_v[v]$, and for each proposition $\pi \in \Pi$ defines $s[\pi] = I_p[\pi]$. $I_v[v]$ is a value in D or nil (undefined), whereas $I_p[\pi] \in B$. An interval $\sigma =< s_0, s_1, \cdots >$ 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 N_0 of non-negative integers and ω, $N_\omega = N_0 \cup \{\omega\}$, and extend the comparison operators, $=, <, \leq$, to N_ω by considering $\omega = \omega$, and for all $i \in N_0, i < \omega$. 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 $< s_i, \ldots, s_j >$ and $\sigma(k)(0 \leq k \preceq |\sigma|)$ denotes $< s_k, ..., s_{|\sigma|} >$. The concatenation of σ with another interval (or empty string) σ' is denoted by $\sigma \cdot \sigma'$. To define the semantics of the projection operator we need an auxiliary operator for intervals. Let $\sigma =< s_0, s_1, \cdots >$ be an interval and $r_1, \ldots, r_h$ be integers $(h \geq 1)$ such that $0 \leq r_1 \leq r_2 \leq \cdots \leq r_h \preceq |\sigma|$. The projection of σ onto $r_1, \ldots, r_h$ is the interval (called projected interval), $\sigma \downarrow (r_1, \ldots, r_h) =< s_{t_1}, s_{t_2}, \ldots, s_{t_l} >$, where $t_1, \ldots, t_l$ is obtained from $r_1, \ldots, r_h$ by deleting all duplicates. For example,

$$< s_0, s_1, s_2, s_3, s_4 > \downarrow (0, 0, 2, 2, 2, 3) =< s_0, s_2, s_3 >$$

An interpretation for a PTL term or formula is a tuple $I = (\sigma, i, k, j)$, where $\sigma =< s_0, s_1, \cdots >$ 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 $I = (\sigma, i, k, j)$ is defined as $I[e]$, by induction on the structure of a term, as shown in Fig.1, where v is a variable and $e_1, \ldots, e_m$ are terms.

$$\mathcal{I}[v] = s_k[v] = I_v^k[v]$$

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

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

$$\mathcal{I}[f(e_1, \ldots, e_m)] = \begin{cases} f(\mathcal{I}[e_1], \ldots, \mathcal{I}[e_m]) & \text{if } \mathcal{I}[e_h] \neq nil \text{ for all } h \\ nil & \text{otherwise} \end{cases}$$

Fig. 1. Interpretation of PTL terms

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

1. $\mathcal{I} \models \pi$ if $s_k[\pi] = I_p^k[\pi] =$ 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 P is a primitive predicate other than = and, for all h, $1 \leq h \leq m, \mathcal{I}[e_h] \neq nil$ and $P(\mathcal{I}[e_1], \ldots, \mathcal{I}[e_m]) =$ 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 in the values assigned to variable v at k.
7. $\mathcal{I} \models \bigcirc p$ if $k < j$ and $(\sigma, i, k+1, j) \models p$.

8. $\mathcal{I} \models (p_1, \ldots, p_m)\, prj\ q$ if there exist integers $k = r_0 \leq r_1 \leq \ldots \leq r_m \leq j$ such that $(\sigma, i, r_0, r_1) \models p_1$, $(\sigma, r_{l-1}, r_{l-1}, r_l) \models p_l$ (for $1 < l \leq m$), and $(\sigma', 0, 0, |\sigma'|) \models q$ for one of the following σ':
   (a) $r_m < j$ and $\sigma' = \sigma\downarrow(r_0, ..., r_m) \cdot \sigma_{(r_{m+1..j)}}$
   (b) $r_m = j$ and $\sigma' = \sigma\downarrow(r_0, \ldots, r_h)$ for some $0 \leq h \leq m$.

A formula p is said to be:

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

The abbreviations $true, false, \wedge, \rightarrow and \leftrightarrow$ are defined as usual. In particular, $true \stackrel{\text{def}}{=} P \vee \neg P$ and $false \stackrel{\text{def}}{=} \neg P \wedge P$ for any formula P. Also some derived formulas are shown in Fig.2.

$$
\begin{array}{llll}
empty \stackrel{\text{def}}{=} \neg \bigcirc true & & more \stackrel{\text{def}}{=} \neg empty & \\
halt(p) \stackrel{\text{def}}{=} \Box(empty \leftrightarrow p) & & keep(p) \stackrel{\text{def}}{=} \Box(\neg empty \rightarrow p) & \\
fin(p) \stackrel{\text{def}}{=} \Box(empty \rightarrow p) & & skip \stackrel{\text{def}}{=} \neg empty & \\
x\circ = e \stackrel{\text{def}}{=} \bigcirc x = e & & x := e \stackrel{\text{def}}{=} skip \wedge x\circ = e & \\
len(0) \stackrel{\text{def}}{=} empty & & len(n) \stackrel{\text{def}}{=} \bigcirc len(n-1)(n > 0) &
\end{array}
$$

Fig. 2. Derived formulas

2.2 Propositional Projection Temporal Logic

Let $Prop$ be a countable set of atomic propositions. The formula of PPTL is given by the following grammar:

$$p ::= \pi \mid \bigcirc p \mid \neg p \mid p_1 \vee p_2 \mid (p_1, \ldots, p_m)\, prj\ p \mid p^+$$

where $\pi \in Prop$, $p_1, \ldots, p_m$ are all well-formed PPTL formulars. A formula is called a state formula if it contains no temporal operators.

Following the definition of Kripke structure, we define a state s over $Prop$ to be a mapping from $Prop$ to $B = \{true, false\}$, $s : Prop \rightarrow B$. We will use $s[\pi]$ to denote the valuation of π at state s. Intervals, interpretation and satisfaction relation can be defined in the same way as in the first order case.

2.3 Modeling, Simulation and Verification Language

The language MSVL with frame [8] technique is an executable subset of PTL and used to model, simulate and verify concurrent systems. The arithmetic expression e and boolean expression b of MSVL are inductively defined as follows:

$$
\begin{array}{l}
e ::= n \mid x \mid \bigcirc x \mid \ominus x \mid e_0\, op\, e_1 (op ::= + \mid - \mid * \mid / \mid mod) \\
b ::= true \mid false \mid e_0 = e_1 \mid e_0 < e_1 \mid \neg b \mid b_0 \wedge b_1
\end{array}
$$

where n is an integer and x is a variable. The elementary statements in MSVL are defined as follows:

Assignment:	$x = e$
P-I-Assignment:	$x \Leftarrow e$
Conditional:	if b then p else $q \stackrel{\text{def}}{=} (b \rightarrow p) \wedge (\neg b \rightarrow q)$
While:	while b do $p \stackrel{\text{def}}{=} (b \wedge p)^* \wedge \Box(empty \rightarrow \neg b)$
Conjunction:	$p \wedge q$
Selection:	$p \vee q$
Next:	$\bigcirc p$
Always:	$\Box p$
Termination:	$empty$
Sequential:	$p; q$
Local variable:	$\exists x : p$
State Frame:	$lbf(x)$
Interval Frame:	$frame(x)$
Parallel:	$p \parallel q \stackrel{\text{def}}{=} p \wedge (q; true) \vee q \wedge (p; true)$
Projection:	$(p_1, \ldots, p_m)$ *prj* q
Await:	$await(b) \stackrel{\text{def}}{=} (frame(x_1) \wedge \cdots \wedge frame(x_h)) \wedge \Box(empty \leftrightarrow b)$ where $x_i \in V_b = \{x \mid x \text{ appears in } b\}$

where x denotes a variable, e stands for an arbitrary arithmetic expression, b a boolean expression, and $p_1, \ldots, p_m$, p and q stand for programs of MSVL. The assignment $x = e$, $x \Leftarrow e$, and $empty$, $lbf(x)$ as well as $frame(x)$ can be regarded as basic statements and the others composite ones.

The assignment $x = e$ means that the value of variable x is equal to the value of expression e. Positive immediate assignment $x \Leftarrow e$ indicates that the value of x is equal to the value of e and the assignment flag, p_x, for variable x is true. Statements $if\ b\ then\ p\ else\ q$ and $while\ b\ do\ p$ are the same as that in the conventional imperative languages. $p \wedge q$ means that p and q are executed concurrently and share all the variables during the mutual execution. $p \vee q$ means p or q are executed. The next statement $\bigcirc p$ means that p holds at the next state while $\Box p$ means that p holds at all the states over the whole interval from now. $empty$ is the termination statement meaning that the current state is the final state of the interval over which the program is executed. The sequence statement $p; q$ means that p is executed from the current state to its termination while q will hold at the final state of p and be executed from that state. The existential quantification $\exists x : p$ intends to hide the variable x within the process p. $lbx(x)$ means the value of x in the current state equals to value of x in the previous state if no assignment to x occurs, while $frame(x)$ indicates that the value of variable x always keeps its old value over an interval if no assignment to x is encountered. Different from the conjunction statement, the parallel statement allows both the processes to specify their own intervals. e.g., $len(2) \| len(3)$ holds but $len(2) \wedge len(3)$ is obviously false. Projection can be thought of as a special parallel computation which is executed on different time scales. The projection $(p_1, \ldots, p_m)$ *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_is are executed. In particular, the sequence of p_is and q may terminate at different time points. Finally, $await(b)$ does not change any variable, but waits until the condition b becomes true, at which point it terminates.

Further, the following derived statements are useful in practice.

Multiple Selection:	$OR_{K=1}^{n} \stackrel{\text{def}}{=} p_1 \vee p_2 \vee \ldots \vee p_n$
Conditional:	if b do $p \stackrel{\text{def}}{=}$ if b do p else empty
When:	when b do $p \stackrel{\text{def}}{=} await(b); p$
Guarded Command:	$b_1 \rightarrow p_n \Box \ldots \Box b_n \rightarrow p_n \stackrel{\text{def}}{=} OR_{k=1}^{n}$ (when b_k do p_k)
Repeat:	repeat p until $c \stackrel{\text{def}}{=} p; while \neg c$ do p

2.4 Unified Model Checking Approach

The idea of unified model checking approach [11] is as follows: modeling the system to be verified by an MSVL program p, and specifying the desired property of the system by a PPTL formula ϕ, to check whether or not the system satisfies the property, we need to prove the validation of

$$p \rightarrow \phi$$

If $p \rightarrow \phi$ is valid, the system satisfies the property, otherwise, the system violates the property. Equivalently, we can check the satisfiabbility of

$$\neg(p \rightarrow \phi) \equiv p \wedge \neg\phi$$

If $p \wedge \neg\phi$ is unsatisfiable, $(p \rightarrow \phi)$ is valid, and the system satisfies the property, otherwise, the system fails to satisfy the property. For each $\sigma \models p \wedge \neg\phi$, σ determines a counterexample that the system violates the property. Accordingly, our model checking approach can be translated to the satisfiability of PTL formulas of the form $p \wedge \neg\phi$, where p is an MSVL program and ϕ is a formula in PPTL. Since both model p and property ϕ are formulas in PTL, we call this model checking approach as unified model checking.

To check the satisfiability of PTL formula $p \wedge \neg\phi$, we construct the NFG (Normal Form Graph) of $p \wedge \neg\phi$. As depicted in Fig.3, initially, we create the root node $p \wedge \neg\phi$, then we rewrite p and $\neg\phi$ into their normal forms respectively. By computing the conjunction of normal forms of p and $\neg\phi$, new nodes ϵ and $p_{fj} \wedge \neg\phi_{fs}$, and edges $(p \wedge \neg\phi, p_{ei} \wedge \neg\phi_{ck}, \epsilon)$ from node $p \wedge \neg\phi$ to ϵ, $(p \wedge \neg\phi, p_{cj} \wedge \neg\phi_{cs}, p_{fj} \wedge \neg\phi_{fs})$ from $p \wedge \neg\phi$ to $p_{fj} \wedge \neg\phi_{fs}$ are created. Further, by dealing with each new created nodes $p_{fj} \wedge \neg\phi_{fs}$ using the same methods as the root nodes $p \wedge \neg\phi$ repeatedly, the NFG of $p \wedge \neg\phi$ can be produced. Thus, it is apparent that each node in the NFG of $p \wedge \neg\phi$ is in the form of $p' \wedge \neg\phi'$, where p' and ϕ' are nodes in the NFGs of p and $\neg\phi$ respectively.

In the NFG of formula $q \equiv p \wedge \neg\phi$, a finite path, $\Pi = < q, q_e, q_1, q_{1e}, \ldots, \epsilon >$, is an alternate sequence of nodes and edges from the root to ϵ node, while an infinite path, $\Pi = < q, q_e, q_1, q_{1e} >$, is an infinite alternate sequence of nodes and edges emanating from the root. Similar to the proof in [12], it can be proved that, the paths in the NFG of q precisely characterize models of q. Thus, if there exists a path in the NFG of q, q is satisfiable, otherwise unsatisfiable.

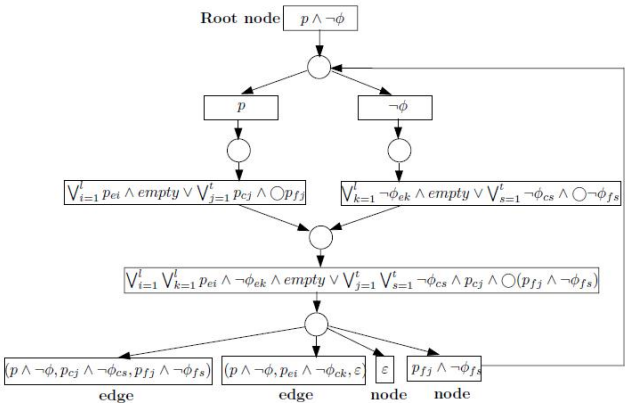

Fig. 3. Constructing NFG of $p \wedge \neg\phi$

3 Translating C Programs into MSVL Programs

In this section, we present the main procedure of transforming a C program into an MSVL program, Fig.4 shows the general process. First of all, the original C program is input into the lexical and syntax analyzers for the analysis. After this process, all useful fragments including identifiers, operators, expressions, and statements in a C program are identified and stored in a specified structure. Finally, we invoke the translation program written in C++ to extract information from the storage structure and to output the translated MSVL program.

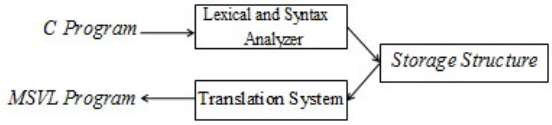

Fig. 4. General Process of the Translation

3.1 Lexical and Syntax Analysis

The first step is to identify the identifiers and statements in the original C program. We accomplish the identification with the help of a common tool named Parser Generator (PG) which integrates Lex and Yacc together. PG is a generator of lexical and syntax analyzers. What we need to do first is to specify the regular expressions for key words (or tokens) and all statements in C program. The regular expressions are defined in two kinds of documents with the extension being .l and .y. Lex source file is a table of regular expressions and corresponding program fragments. It will be translated into a program which reads an input C program stream, copying it to an output stream and partitioning the input stream into tokens which match our given expressions. Yacc source

file specifies the structures of the input, together with code to be invoked as each such structure is recognized, and Yacc turns such a specification into a subroutine. As we can see, based on these two kinds of specific documents, PG will automatically generate an analysis program in C++, which implements the identification of original C programs.

The key step of lexical and syntax analysis is the specification of regular expressions for basic tokens and statements in C programs. A C program normally consists of a list of elementary statements, and for each normative statement, it may contain keywords, basic expressions, sub-statements or a block of statements.

Lex source file specifies the regular expressions of basic constructs in C language, such as variables, characters set, data types, constants, keywords (reserved words in C language), identifiers, arrays and so on. Yacc source file defines regular expressions for elementary C statements, as well as the corresponding handle functions when such statements are matched. Fig.5 shows the basic process of lexical and syntax analysis. At first, source files $lexer.l$ and $parser.y$ are input into PG process. After the analysis, executable programs in C++ are generated. By compiling the programs, we finally obtain an analyzer for C programs.

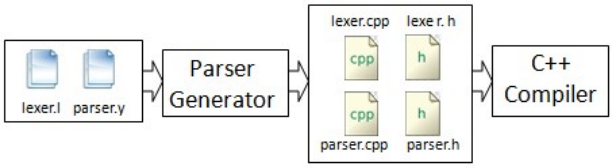

Fig. 5. Process of Lexical and Syntax Analysis

3.2 Storage Structure

To store information, we formalize two classes: one for expressions and statements named AstNode, and the other one for statement blocks named StmtBlock. Member variables of AstNode and StmtBlock are shown in Fig.6 and Fig.7 respectively.

As shown in Fig.6, class AstNode contains four primary member variables: AstNode $*left$, AstNode $*right$, StmtBlock $*block$ and AstNodeType $type$. $left$ and $right$ are used to store basic expressions or sub-statements that may appear in elementary C statements. $type$ is an enumeration variable used to indicate the type of statements or expressions. For instance, *ANT_FOR* represents "for-statement" and *ANT_ADD* stands for plus(+) operation. The remaining member variables such as string str_1 and int int_1 are auxiliaries used to store the name of a variable or the value of an integer variable. Each statement in C programs is related to an instance of class AstNode.

Fig.7 shows the structure of class StmtBlock, it contains two primary member variables: AstNode $*stmt$ and StmtBlock $*block$. Actually, a StmtBlock instance can be regarded as a list of AstNode instances which relates to a statement block.

$$\textbf{AstNode}\begin{cases} AstNode * left \\ AstNode * right \\ StmtBlock * block \\ AstNodeType \quad type \\ string \quad str1 \\ int \quad int1 \end{cases}$$

Fig. 6. Class *AstNode* Structure

$$\textbf{stmtblock}\begin{cases} AstNode * stmt \\ StmtBlock * next \end{cases}$$

Fig. 7. Class *StmtBlock* Struture

Member functions are also needed to manipulate the member variables. Here we mainly introduce function $allco()$ since it is frequently used.

- For class AstNode, $alloc()$ generates an AstNode instance dynamically, meanwhile, the function needs three formal parameters: AstNode $*left$, AstNode $*right$ and AstNodeType $type$.
- For class StmtBlock, $alloc()$ generates a StmtBlock instance dynamically. The function needs only one formal parameter: AstNode $*p$, which points to the head of the AstNode list.

Whenever a regular expression in Yacc source file is matched, corresponding handle functions such as $alloc()$ will be invoked, therefore, statement fragments can be stored in the newborn class instance. Functions dealing with setting and getting member variables are also needed, we omit the details here.

As to the storage of the whole C program, we make use of a container in STL(Standard Template Library) named $vector$, which acts as a dynamic array. A variety of functions are encapsulated in $vector$, so we can call them directly, such as $push\_back()$ and $pop\_back()$. $stmt$ is a container we defined to store all statements in a C program. To store a C fragment in container $stmt$, we only need to call function $stmt.push\_back()$.

3.3 Translation

3.3.1 Translation Algorithm

Algorithm TransForm is formalized to realize the translation. Given a C program fragment S as input, we first check the type of S since different types invoke different methods. The pseudo code of algorithm TranForm are shown in Table.1. In the algorithm, GetStmt is used to transform a C statement into an MSVL statement, while GetExpr will transform a basic expression into an MSVL expression. GetBlock aims at the transforming of a statement block. The general process of the algorithm is given below:

- Input required C fragments into the translation procedure
- Specific methods will be invoked
- These methods translate C fragments and output MSVL fragments

Table 1. Algorithm for translating C fragments to MSVL fragments

Function TransForm(S)
/*precondition: S is an elementary fragment of C program */
/*postcondition: Transform(S) computes an equivalent MSVL fragment*/
begin function
case
S is a statement: **return** GetStmt(S);
S is an expression: **return** GetExpr(S);
S is a statement block: **return** GetBlock(S);
end case
end function

In algorithm GetStmt, basic statements such as integer declaration, if statement, while statement, for statement, printf and scanf need to be considered. Intuitively, ";" represents an empty statement, "exp;" stands for an expression statement and "x=exp;" is an assignment statement. However, in order to improve the efficiency of GetStmt, some auxiliary methods such as GetIfStmt and GetDeclaration are also required. The algorithm is straightforward which includes all cases and uses the corresponding rules to transform related fragments as shown in Table.5 and Table.6.

Table 2. Algorithm for translating a C statement to an MSVL statement

Function GetStmt(S)
/*precondition: S is an elementary C statement */
/*postcondition: GetStmt(S) computes an equivalent MSVL statement*/
/* exp is a standard expression, $block$ is a statement block, s represents a complete statement, x is a variable, $parameter$ stands for a string */
begin function
case
S is ; **return** ;
S is exp; **return** exp;
S is x=exp; **return** x:=$exp\ and\ skip$;
S is $if\text{-}statement$: **return** GetIfStmt(S);
S is $while(exp)\{block\}$: **return** $while(exp)\{block\}$;
S is $int\text{-}declaration\text{-}statement$: **return** GetDeclaration(S);
S is $for(s\ exp_1;exp_2)\{block\}$: **return** $s\ while(exp_1)\{block;exp_2;\}$;
S is $printf(\text{“}parameter\text{”},exp)$: **return** $output(exp)$;
S is $scanf(\text{“}parameter\text{”},exp)$: **return** $input(exp)$;
end case
end function

Table 3. Algorithm for translating C expressions to MSVL expressions

Function GetExpr(S)
/*precondition: S is an elementary C expression */
/*postcondition: GetExpr(S) computes an equivalent MSVL expression*/
/*x and y are standard expressions*/
/*e represents a constant, a variable or an identifier */

begin function
case
S is e: **return** e
S is $x\circledast$ ($\circledast$ =[++|- -]): **return** x:=x+1 $and\ skip \mid x$:=x-1 $and\ skip$
S is x=y: **return** x:=y $and\ skip$
S is x==y: **return** x=y
S is x[+,-,*,/,%,!=]y: **return** x[+,-,*,/,%,!=]y
S is x*=y (*=[<|>]): **return** x=y or x*y
S is x[+|-|*|/|%]=y: **return** x:=x[+|-|*|/|%]y $and\ skip$
S is x&&y: **return** x and y
S is $x||y$: **return** x or y
S is x,y: **return** x,y
S is (x): **return** (x)
end case
end function

As to algorithm GetExpr given in Table.3, elementary C expressions are considered, such as identifier, constant, arithmetic expression, logical expression, relational expression, bracket expression and so on. Table.4 displays the transformation rules of algorithm GetBlock. Since a statement block is a list of statements linked by pointers, to transform a state block we can repeatedly call GetStmt(S) as long as S is not $null$. $next(S)$ in algorithm GetBlock means that S points to the next statement.

Table 4. Algorithm for translating a statement block

Function GetBlock(S)
/*precondition: S is an elementary C statement block */
/*postcondition: GetBlock(S) computes an equivalent MSVL block*/

begin function

repeat{GetStmt(S); S=$next(S)$;} **until** (S=null;)

end function

Table 5. Algorithm for translating $if\text{-}statement$

Function getIfStmt(S) /*precondition: S is an elementary if-statement in C language */ /*postcondition: getIfStmt(S) computes an equivalent MSVL if-statement*/ /*exp is an standard expression and $block$ is a statement block*/
begin function **case** S is $if(exp)s$: return $if(exp)then\ \{s\}$ S is $if(exp)\{block\}$: return $if(exp)then\{block\}$ S is $if(exp)\{block_1\}else\{block_2\}$: return $if(exp)then\{block_1\}$ $else\{block_2\}$ S is $if(exp_1)\{block_1\}else\ if(exp_2)\{block_2\}$: return $if(exp_1)then$ $\{block_1\}else\ if(exp_2)\{block_2\}$ **end case** **end function**

Table 6. Algorithm for Translating $int\text{-}declaration\text{-}statement$

Function getDeclaration(S) /*precondition: S is an elementary int-declaration-statement in C language */ /*postcondition: getDeclaration(S) computes an equivalent MSVL int-declaration-statement*/ /*var_list represents a list of variables connected by "," */ /*exp_list represents a list of expression connected by ","*/ /*e_i is one of the expressions that appear in exp_list*/
begin function **case** S is $int\ x$;: return $int\ x$; S is $int\ var\_list$;: return $int\ var\_list$; S is $int\ x$=exp;: return $int\ x$; x:=$exp\ and\ skip$; S is $int\ exp\_list$;: return getDeclaration(e_i); ($0<i<=len(exp\_list)$) **end case** **end function**

3.3.2 Implementation

We now focus on the implementations of algorithm TransForm, which includes $GetStmt()$, $GetExpr()$ and $GetBlock()$. The algorithm has two kinds of input parameters: $AstNode$, $StmtBlock$, and different parameters invoke different methods. Since $GetStmt()$ and $GetExpr()$ have much in common, we omit the details for $GetExpr()$ here. Brief descriptions of $GetStmt()$ and $GetBlock()$ are shown in Fig.8 and Fig.9. As we can see, method $Switch$ is the core algorithm for implementations. It generates

Method: $GetStmt()$
Parameter Type: $AstNode$
Function: Realize the translation and output of elementary statements
Process:

- $GetType()$ is called to get $type$, which specifies the type of a statement.
- Call $Switch(type)$ to realize the transformation and output

Fig. 8. GetStmt()

Method: $GetBlock()$
Parameter Type: $StmtBlcok$
Function: Realize the translation and output of a statement block
Process:

- A repeated calling of $GetStmt()$

Fig. 9. GetBlock()

different results according to parameter $type$. $type$ is more like a statement label which informs $Switch$ of which transformation rule should be effective .

So far, the transformation from basic C programs to MSVL programs is almost done. However, some problems are worth paying attention in order to make sure that the final output fits the MSVL interpreter well. What we need to point out here is an obvious difference between C and MSVL: in C programs, the last statement in a block must ends with a semicolon(;), however, in MSVL the semicolon has been saved. To take both situations into consideration, a method named $GetStmt\_last()$ is designed to cope with this problem. $GetStmt\_last()$ is totally as the same as $GetStmt()$ except that the former gets rid of the semicolon at the end of a statement.

4 A Case Study

In this section, we will give a basic but typical C program which consists of a number of elementary C statements. Then, by employing our translation algorithm, we can get an equivalent MSVL program.

4.1 Greatest Common Divisor and Lowest Common Multiple

There are many ways to find the Greatest Common Divisor (GCD) and the Lowest Common Multiple (LCM) of two positive integers. Here we use Euclidean Algorithm (Euclid's Algorithm) to find GCD and a particular formula to calculate LCM of two positive integers.

4.1.1 Euclidean Algorithm

This algorithm finds GCD by performing repeated division starting from the two numbers we want to find out the GCD until we get a remainder of 0. Below are the steps to compute GCD of positive integers, 12 and 8, by using Euclid's algorithm.

- Divide the larger number by the small one. In this case we divide 12 by 8 to get a quotient of 1 and remainder of 4.
- Next we divide the smaller number (i.e. 8) by the remainder from the last division (i.e. 4). So 8 is divided by 4, and we get a quotient of 2 and remainder of 0.
- Since we already get a remainder of 0, the last number that we used to divide is the GCD, i.e 4.

Implementations of the algorithm may be expressed in pseudo code. For example, the division-based version may be programmed as below:

```
function gcd(a, b)
while b != 0
t := b
b := a mod b
a := t
return a
```

4.1.2 A Formula to Find LCM

We can use the following formula to calculate LCM of positive integer a and b if we already know $GCD(a, b)$.

$$\text{LCM(a,b)} = \frac{\text{a} \times \text{b}}{\text{GCD(a,b)}}$$

Fig. 10. Formula for Calculating LCM

4.2 Translation from C to MSVL

The implementation code in Fig.11 shows how to find GCD and LCM of two positive integers: a and b. It is based on the algorithm we mentioned in section 4.1. Note that the code contains elementary C statements, such as variable declaration, assignment statement, arithmetic expression, if statement, while statement, printf statement and scanf statement. After employing the transformation algorithm to the original C program, an equivalent MSVL program is generated as shown in Fig.12.

To test the correctness of the MSVL program, as illustrated in Fig.13, we use two positive integers: 256 and 60. According to the execution result, GCD(256,60) equals to 4 and LCM(256,60) equals to 3840, which is apparently correct.

```
#include<stdio.h>
int main()
{
  int num1,num2,a,b,t;
  printf("please input two numbers:");
  scanf("%d %d",&a,&b);
  if(a<b){
       t=a;
       a=b;
       b=t;}
  num1=a;num2=b;
  while(num2!=0){
       t=num2;
       num2=num1%num2;
       num1=t;}
  printf("GCD:%d\n",num1);
  printf("LCM:%d\n",(a*b)/num1);
  return 0;
}
```

Fig. 11. C Program

```
frame(num1,num2,a,b,t) and skip;
int num1,num2,a,b,t;
output("please input two numbers:") and skip;
input(a) and input(b)and skip ;
if ( a<b ) then{
t:= a and skip;
a:= b and skip;
b:= t and skip
} and skip;
num1:= a and skip;
num2:= b and skip;
while(num2!=0){
t:= num2 and skip;
num2:= num1 % num2 and skip;
num1:= t and skip
};
output ("GCD:",num1) and skip ;
output ("LCM:", (a*b)/num1) and skip
```

Fig. 12. MSVL Program

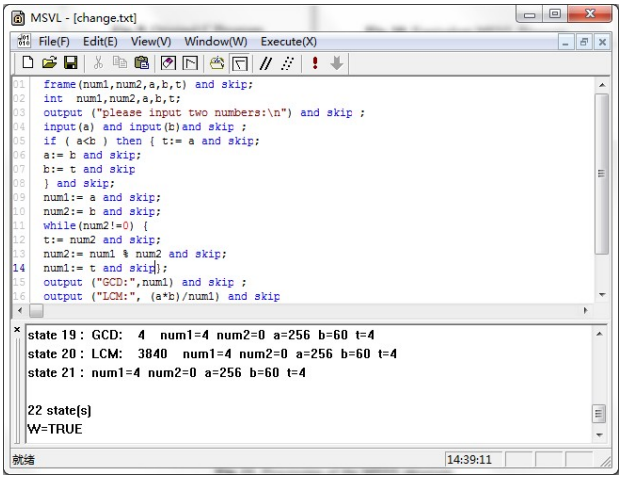

Fig. 13. Executing of the Final MSVL program

4.3 Verification with MSVL

After the translation, we can apply the existing unified model checking approach to the MSVL program. Since the C program and the MSVL program are equivalent, the validity of the MSVL program also indicates the validity of the original C program. As to our example, two positive integers are 256 and 60. It is easy to find out that the property "final GCD either less than 60 or equals to 60" should always hold. By employing propositions p and q to denote $gcd{<}small$ and $gcd{=}small$ respectively,

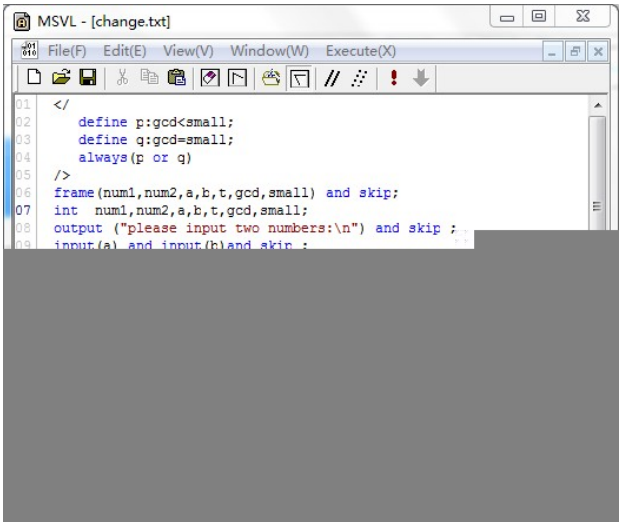

Fig. 14. Verification Result

this property can be specified by $\Box(p \vee q)$ in PPTL. Under the verification mode of MSVL, we add the following code

```
</
  define p:gcd<small;
  define q:gcd=small;
  always(p or q)
/>
```

to the beginning of the MSVL program. In this property, gcd represents the final GCD of the two positive integers, $small$ is the smaller integer of the two positive integers. After running the program, an empty NFG with no edge is produced as shown in Fig.14. Hence,the formula is unsatisfiable, and the system satisfies the property.

5 Conclusion

In this paper, we present a translator from C programs to MSVL programs. Therefore, by transforming a C program into an equivalent MSVL program, we can employ the existing techniques to verify the correctness of C programs. This enables us to translate the problem of checking whether or not the C program satisfies the property to the problem of checking the satisfiability of MSVL programs.

However, we just realize the transformation from basic C programs to MSVL programs at the moment. For some complex C programs, further investigations are still needed. Currently, the translator is merely a prototype, and lots of efforts are needed to improve it. In addition, to examine our method, several big case studies are required in the near future.

References

1. Ostroff, J.S.: Verification of safety critical systems using TTM/RTTL. In: Huizing, C., de Bakker, J.W., Rozenberg, G., de Roever, W.-P. (eds.) REX 1991. LNCS, vol. 600, pp. 573–602. Springer, Heidelberg (1992)
2. Yang, M., Wang, Z., Pu, G., Qin, S., Gu, B., He, J.: The Stochastic Semantics and Verification for Periodic Control Systems. Science China: Information Sciences 55(12), 1–19 (2012)
3. Qin, S., Luo, C., Chin, W.-N., He, G.: Automatically Refining Partial Specifications for Program Verification. In: Butler, M., Schulte, W. (eds.) FM 2011. LNCS, vol. 6664, pp. 369–385. Springer, Heidelberg (2011)
4. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. 2(4), 255–299 (1990)
5. Ghezzi, C., Mandrioli, D., Morzenti, A.: Specifying real-time properties with metric temporal logic. J. Syst. Softw. 12(2), 107–123 (1990)
6. Jahanian, F., Mok, A.K.: Safety analysis of timing properties in real-time systems. IEEE Trans. Softw. Eng. SE-12(9), 890–904 (1986)
7. Duan, Z.: An Extended Interval Temporal Logic and A Framing Technique for Temporal Logic Programming. PhD Thesis, University of Newcastle upon Tyne (1996)
8. Duan, Z.: Temporal Logic and Temporal Logic Programming. Science Press, Beijing (2006)
9. Alur, R., Henzinger, T.A.: A really temporal logic. In: Proceedings of the 30th IEEE Conference on Foundations of Computer Science. IEEE Computer Society Press, Los Alamitos (1989)
10. Melliar-Smith, P.M.: Extending interval logic to real time systems. In: Banieqbal, B., Pnueli, A., Barringer, H. (eds.) Temporal Logic in Specification. LNCS, vol. 398, pp. 224–242. Springer, Heidelberg (1989)
11. 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)
12. Duan, Z., Tian, C., Zhang, L.: A decision procedure for propositional projection temporal logic with infinite models. Acta Informatica 45(1), 43–78 (2008)