State Space Reduction Strategies for Model Checking Concurrent C Programs

Amira Methni, Belgacem Ben Hedia, Matthieu Lemerre,
CEA, LIST, Centre de Saclay,
PC172, 91191, Gif-sur-Yvette, FRANCE
{*amira.methni,belgacem.ben-hedia, matthieu.lemerre*}*@cea.fr*

Serge Haddad
LSV, ENS Cachan, CNRS
& INRIA, France
haddad@lsv.ens-cachan.fr

Kamel Barkaoui
CEDRIC Lab, CNAM,
Paris, France
kamel.barkaoui@cnam.fr

Model checking is an effective technique for uncovering subtle errors in concurrent systems. Unfortunately, the state space explosion is the main bottleneck in model checking tools. Here we propose a state space reduction technique for model checking concurrent programs written in C. The reduction technique consists in an analysis phase, which defines an approximate *agglomeration predicate*. This latter states whether a statement can be agglomerated or not. We implement this predicate using a syntactic analysis, as well as a semantic analysis based on abstract interpretation. We show the usefulness of using agglomeration technique to reduce the state space, as well as to generate an abstract TLA+ specification from a C program.

Model checking, TLA, State space reduction, Agglomeration predicate.

1. INTRODUCTION

Model checking is an attractive formal verification technique because it is automatic. It offers extensive and thorough coverage by considering all possible behaviors of a system, unlike traditional testing methods. Given a set of properties expressed in a temporal logic and a model, model-checking automatically analyzes the state space of the model and checks whether the model satisfies the properties (Clarke et al. 1999). However, the main obstacle of model checking is the state explosion problem and concurrency is a major contributor to this problem.

Many solutions have already been investigated for reducing the complexity of model checking. For instance, by getting a simpler model from the original one using abstraction technique (Clarke et al. 1994), or by using on-the-fly model checking to eliminate part of the search to the automaton representing the (negation of the) checked property (Fernandez et al. 1992).

1.1. Contribution

In this paper, we present a state space reduction technique for model checking concurrent programs written in a low level language. We apply this technique to the verification of C programs by an explicit model checker. We use TLA+ (Lamport 1994) as a formal specification language for our concurrent C programs and we base ourselves on previous work reported in (Methni et al. 2015). The reduction technique is based on an analysis phase, which defines an approximate *agglomeration predicate*. This latter states whether a statement can be agglomerated or not. We implement this predicate using a syntactic analysis, as well as a semantic analysis based on abstract interpretation of C code. The particularity of our method is that we apply the reduction technique during the generation of TLA+ code and by using the abstract interpretation technique. We show the usefulness of using this technique to reduce the state space during the verification of C programs, as well as to generate an abstract TLA+ specification from a C program.

1.2. Outline

The rest of the paper is organized as follows. We give an overview of TLA+ in Section 2. Section 3 presents how we specify the semantics of C in TLA+. Section 4 describes the reduction technique and how we implement it on C programs. Experimental results are presented in Section 5. We discuss related work in Section 6. Section 7 concludes and illustrates future research directions.

2. OVERVIEW OF TLA

TLA+ (Lamport 2002) is the specification language of the Temporal Logic of Actions (TLA). TLA is

$$\begin{array}{lcl}
\langle formula \rangle & \triangleq & \langle predicate \rangle \mid \Box[\langle action \rangle]_{\langle state\ function \rangle} \mid \neg \langle formula \rangle \\
& & \mid \langle formula \rangle \wedge \langle formula \rangle \mid \Box \langle formula \rangle \\
\langle action \rangle & \triangleq & \text{boolean valued expression containing constant symbols, variables,} \\
& & \text{and primed variables} \\
\langle predicate \rangle & \triangleq & \langle formula \rangle \text{ with no primed variables} \mid \text{ENABLED } \langle action \rangle \\
\langle state\ function \rangle & \triangleq & \text{nonboolean expression containing constant symbols and variables}
\end{array}$$

Figure 1: *TLA syntax (Lamport 2002)*

a variant of linear temporal logic introduced by (Lamport 1994) for specifying and reasoning about concurrent systems. Readers interested in a more detailed presentation of TLA+ can refer to Lamport's book (Lamport 2002).

TLA+ specifies a system by describing its possible behaviors. A *behavior* is an infinite sequence of states. A *state* is an assignment of values to variables. A *state function* is a nonboolean expression built from constants, variables and constant operators and it assigns a value to each state. For example, $y + 3$ is a state function that assigns to state s the value 3 plus the value that s assigns to the variable y. An *action* is a boolean expression containing constants, variables and primed variables (adorned with "′" operator). Unprimed variables refer to variable values in the current state and primed variables refer to their values in the next-state. Thus, an action represents a relation between old states and new states. A *state predicate* (or predicate for short) is an action with no primed variables.

The syntax of TLA is given in Figure 1 (the symbol $\triangleq$ means *equal by definition*). TLA+ formulas are built up from actions and predicates using boolean operators ($\neg$ and $\wedge$ and others that can be derived from these two), quantification over logical variables ($\forall, \exists$), and the unary temporal operator $\Box$ (*always*) of the linear temporal logic (Manna and Pnueli 1992).

The predicate "ENABLED $\mathcal{A}$", where $\mathcal{A}$ is an action, is defined to be true in a state s iff there exists some state t such that the pair of states $\langle s, t \rangle$ satisfies $\mathcal{A}$. The formula $[\mathcal{A}]_{vars}$, where $\mathcal{A}$ is an action and $vars$ the tuple of all system variables, is equal to $(\mathcal{A} \vee (vars' = vars))$ where $vars'$ is the expression obtained by priming all variables in $vars$. It asserts that every step (pair of successive states) is either an $\mathcal{A}$ step or else leaves the values of all variables $vars$ unchanged. TLA+ defines the abbreviation "UNCHANGED $vars$" to denote that $vars' = vars$.

While TLA+ permits a variety of specification styles, the specification that we use is defined by:

$$Spec \triangleq Init \wedge \Box[Next]_{vars} \wedge Fairness \qquad (1)$$

where:

- $Init$ is a state predicate describing the possible initial states by assigning values to all system variables,
- $Next$ is an action representing the program's next-state relation,
- $vars$ is the tuple of all variables,
- $Fairness$ is an optional formula representing weak or strong assumptions about the execution of actions.

Formula $Spec$ is true of a behavior σ iff $Init$ is true of the first state of σ and every step of σ is either a $Next$ step or a "stuttering step", in which none of the specified variables change their values, and $Fairness$ holds. The behaviors satisfying the specification formula given by Equation (1) are the ones that represent correct behaviors of the system, where a behavior represents a conceivable history of a universe that may contain the system.

The TLA+ formula $Spec \Rightarrow \phi$ is valid when the model represented by $Spec$ satisfies the property ϕ, or implements the model ϕ.

TLA+ has an explicit model checker called TLC that can be used to check the validity of safety and liveness properties. TLC handles specifications that have the standard form of the formula (1). For this reason, we only use specification formula of the form of Equation (1). TLC requires a configuration file which defines the finite-state instance to analyze. It begins by generating all states satisfying the initial predicate $Init$. Then, it generates every possible next-state t such that the pair of states $\langle s, t \rangle$ satisfies $Next$ and the $Fairness$ constraints, looking for a state where an invariant is violated. Finally, it checks temporal properties over the state space.

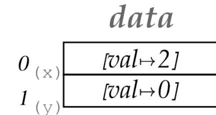
3. TRANSLATION FROM C TO TLA+

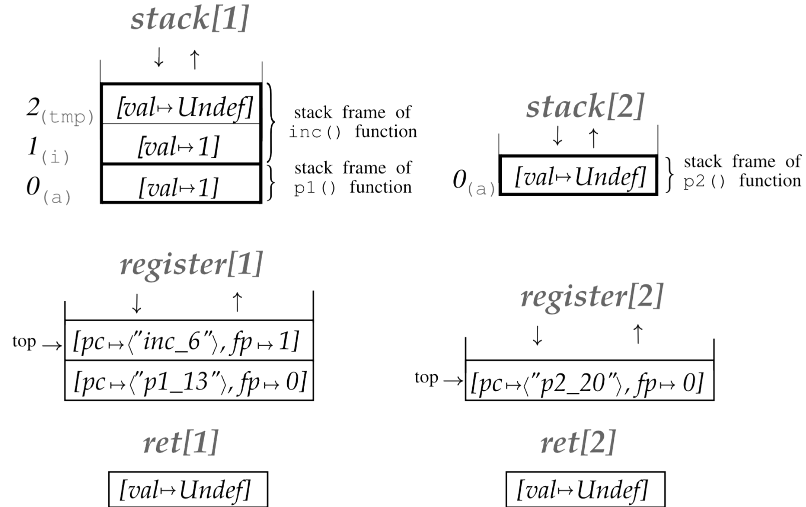
Our approach to checking a concurrent C program is to first translate it into a TLA+ specification, to which the TLA+ tools can be applied. In what follows, we briefly present how we specify the semantics of C in

```c
int x = 3;
int y = 0;

int inc(int i)
{
  int tmp;
  tmp = i+1;
  return tmp;
}

void p1(){
  int a = 1;
  x = inc(a);
  a = x;
  return;
}

void p2(){
  int a;
  x = x - 1;
  x = 3;
  y = 0;
}
```

(a) Code C source

$Addr_x = [loc \mapsto "data", offs \mapsto 0]$
$Addr_y = [loc \mapsto "data", offs \mapsto 1]$
$Addr_p1_a = [loc \mapsto "data", offs \mapsto 1]$
$Addr_p2_b = [loc \mapsto "data", offs \mapsto 1]$
$Addr_inc_param_i = [loc \mapsto "stack", offs \mapsto 0]$
$Addr_inc_tmp = [loc \mapsto "stack", offs \mapsto 1]$





***(b)** Memory layout*

***Figure 2:** Example of a C code in which one process (with id equals to 1) executes `p1()` function and the second one executes `p2()`. The top of the $stack[1]$ indicates that process 1 is executing the statement with label 6 of `inc()` function. $Undef$ represents an undefined value such as the value of an uninitialized variable.*

TLA+ by describing the memory layout considered and how we model the control flow of a C program.

3.1. Memory Layout

C file is parsed and normalized according to CIL (C Intermediate Language) (Necula et al. 2002) which transforms complicated constructs of C into simpler ones. This transformation makes programs more amenable to analysis and transformation. According to the Abstract Syntax Tree (AST) of the C program, C2TLA+ generates automatically a TLA+ specification according to a set of translation rules detailed in our previous work (Methni et al. 2015).

In C2TLA+, a concurrent program consists of many interleaved sequences of operations called *processes*, corresponding to threads in C. Each process has a unique identifier id. The set of all processes is determined by the TLA+ constant $ProcSet$.

Figure 2 presents a C program and the content of the memory as modeled by C2TLA+. We consider that the C code is executed by two processes. One process executes `p1()` function and the other one executes `p2()` function.

The memory is separated into four areas that do not overlap:

- a shared memory called $data$ that stores global (and static) variables. In the example of Figure 2a, the `x` variable is shared by the two processes.
- a local memory for each process, called $stack$ and stores local variables and function parameters. The memory $stack[id]$ specifies the local memory of process id and is composed of stack frames. Each stack frame corresponds to a call to a function. In the example of Figure 2a, $stack[1]$ is composed of two stack frames, one of `p1()` function and one of `inc()` function. When a function call terminates, its stack frame is removed.
- a local memory for each process called $register$ modeled as a sequence and stores the program counter of each process. The head of this sequence contains the statement being currently executed by the process id.
- a local memory called ret which contains values to be returned by processes.

The memory is modeled in TLA+ by a variable called $memory$. It is a record whose four fields represent the four memory areas. The global memory $data$ behaves like an array of values, whereas $stack$ and $register$ behave like a FIFO (First In, First Out) queues. Access to those memory areas is addressed using offsets. So, a memory address is a couple $[loc{,}offs]$ of memory location loc, ($data$ or $stack$ area) and an offset $offs$ in this location. For instance, $Addr_x$ specified in Figure 2b defines the memory address of `x` variable.

The main operations that manage the memory are $load()$ and $store()$:

- $load()$ is the function that given the current state of memory mem and a memory address $addr$ (in the form of $[loc, offs]$), returns the value stored at the address $addr$ in the memory mem,
- $store()$ is the function that given the current state of memory mem, a memory address $addr$ and a value val, returns the new copy of the memory after storing the value val at the memory address $addr$.

3.2. Specifying the C control-flow

Each C statement i is identified in C2TLA+ by a label assigned by CIL and is modeled by a TLA+ function, noted $stmt_i()$, which takes as arguments the process identifier id and the memory mem, and returns the new content of the memory after executing the statement.

Each $stmt_i()$ updates the program counter $register$ of the process id and may change the content of mem, $stack$, and/or ret memory areas depending on the type of the statement (assignment, jump statements, etc.). For instance, the statement on line 20 is translated into the TLA+ function $p2_20(id, mem)$ defined as follows:

$$
\begin{array}{l}
p2_20(id, mem) \triangleq \\
\quad \text{LET } mcopy \triangleq load(id, Addr_x, [val \mapsto 3]) \text{ IN} \\
\quad\; [data \mapsto mcopy.mem, stack \rightarrow mcopy.stack, \\
\quad\; register \mapsto [mem.register \text{ EXCEPT } ![id] = \\
\quad\; \langle [pc \mapsto \langle \text{“}p2_21\text{”} \rangle, fp \mapsto Head(mem.register[id]).fp] \rangle \\
\qquad\quad \circ\, Tail(mem.register[id]), \\
\quad\; ret \mapsto mem.ret]
\end{array}
$$

The definition of $p2_20()$ function uses the TLA+ construct LET/IN to define a temporary variable that stores the value of the memory after affecting the value 3 to the memory address $Addr_x$. The symbol $\circ$ defines the concatenation operator for TLA+ sequences. $Head(s)$ is a TLA+ function that returns the head of the sequence s and $Tail(s)$ returns the tail of the sequence s. Then, the $register[id]$ is updated by the label value of the successor statement given by the control flow graph (CFG) of the C program (provided by CIL).

The control flow of the C program in C2TLA+ is ensured by the $dispatch()$ function. For the example of Figure 2a, this function is defined as follows:

$$
\begin{array}{l}
dispatch(id, mem) \triangleq \\
\quad \text{CASE } Head(mem.register[id]).pc = \text{”}inc_6\text{”} \\
\qquad\quad \rightarrow inc_6(id, mem) \\
\qquad \Box\; Head(mem.register[id]).pc = \text{”}inc_7\text{”} \\
\qquad\quad \rightarrow inc_7(id, mem) \\
\qquad \Box\; Head(mem.register[id]).pc = \text{”}p1_11\text{”} \\
\qquad\quad \rightarrow p1_11(id, mem) \\
\qquad \Box\; Head(mem.register[id]).pc = \text{”}p1_12\text{”} \\
\qquad\quad \rightarrow p1_12(id, mem) \\
\qquad \Box\; Head(mem.register[id]).pc = \text{”}p1_13\text{”} \\
\qquad\quad \rightarrow p1_13(id, mem) \\
\qquad \Box\; Head(mem.register[id]).pc = \text{”}p1_14\text{”} \\
\qquad\quad \rightarrow p1_14(id, mem) \\
\qquad \Box\; Head(mem.register[id]).pc = \text{”}p2_19\text{”} \\
\qquad\quad \rightarrow p2_19(id, mem) \\
\qquad \Box\; Head(mem.register[id]).pc = \text{”}p2_20\text{”} \\
\qquad\quad \rightarrow p2_20(id, mem) \\
\qquad \Box\; Head(mem.register[id]).pc = \text{”}p2_21\text{”} \\
\qquad\quad \rightarrow p2_21(id, mem) \\
\qquad \Box\; \text{OTHER } \rightarrow mem
\end{array}
$$

The $dispatch()$ function calls, according to the value of the pc field contained at the top the process register (determined by the expression $Head(mem.register[id]).pc$), the corresponding TLA+ function to execute, i. e., the C instruction to execute.

The C program is thus simulated by the $Spec$ formula given by equation (1). The $Init$ predicate specifies the initial values of the memory and the $Next$ action is defined as follows:

$$
\begin{array}{l}
Next \triangleq \\
\quad \lor\; \exists\, id \in ProcSet : \\
\qquad\quad \land\; memory.regsiter[id] \neq \langle\rangle \\
\qquad\quad \land\; memory' = dispatch(id, mem) \\
\quad \lor\; \forall\, id \in ProcSet : \\
\qquad\quad \land\; memory.regsiter[id] = \langle\rangle \\
\qquad\quad \land\; \text{UNCHANGED } memory
\end{array}
$$

It states that one of the processes that has not finished execution (its $register[id]$ is not empty) is nondeterministically chosen to execute one action until all processes finish execution, i. e., all registers become empty. Executing an action consists in calling $dispatch()$ function. For example, when $Head(mem.register[id]).pc$ equals to ”inc_6”, calling the function $inc_6(id, mem)$ will update the value of $stack[id]$ (as `tmp` is stored in the local memory) as well as the top of $register[id]$. As the $register[id]$ is

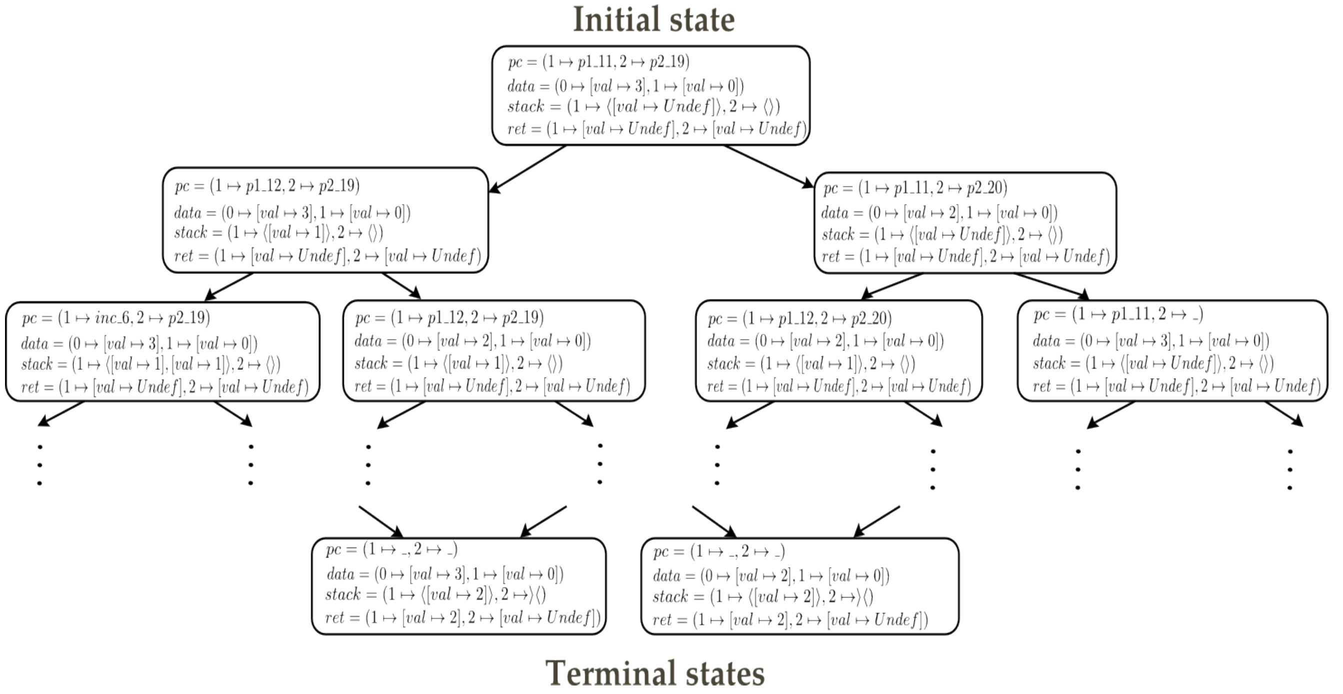


Figure 3: *Example of a state space*

still not empty, the control flow is thus passed to the successor statement.

The behavior of the C program modeled by the $Spec$ formula can be given in terms of a *state transition system*.

Definition 1. A state transition system is a 3-tuple $T = (Q, I, \delta)$ given by

- a finite set of states Q,
- a set $I \subseteq Q$ of initial states, specified by the $Init$ predicate,
- a transition relation $\delta \subseteq Q \times Q$ that links two states. This latter corresponds to satisfying the predicate $Next$.

The state transition system encodes the state space of the corresponding TLA+ specification of the C program.

Figure 3 illustrates the state space of the corresponding TLA+ specification of the C code given in Figure 2a. It consists in all the possible interleaving of process execution. In order to simplify, we represent only the content of pc field contained at the top of the $register[1]$ and $register[2]$ memories. Each state of the graph matches a valuation of $memory$ variable, i. e., its four fields.

3.3. Process Synchronization

All processes interact with each other through the shared memory $data$. Concurrent access to this latter is ensured via synchronization mechanism. There are many different ways to implement concurrency synchronization in C. For instance, by using locks and semaphores, or by providing low level hardware instructions (e. g., *test-and-set* and *compare-and-swap*). To support synchronization mechanism, generated TLA+ specifications by C2TLA+ can be completed with manually written TLA+ specifications to provide concurrency primitives and atomic instructions. More detailed information about integrating synchronization primitives in TLA+ specifications can be found in our previous work (Methni et al. 2015).

4. APPLYING REDUCTION ON C PROGRAMS

The process of generating an optimized TLA+ specification is illustrated in Figure 4. To apply reduction on C programs, it is necessary to define the agglomeration predicate. The C program is first analyzed. This analysis phase defines an approximate *agglomeration predicate* which takes as argument a C statement and returns true or false depending on whether the statement can be agglomerated or not. This predicate can be *safe* or *unsafe*. The meaning of *safe predicate* depends on how the analysis is performed.

- The predicate is said safe when the analysis is a safe approximation. Its definition is as follows:
  - if the statement does not modify the shared memory, the predicate returns true,

  - if the statement modifies the shared memory or it is *unknown*, it returns false.

  The *unknown* predicate states that we have no idea if the statement modifies the memory or not.

- The predicate is called unsafe when a statement is agglomerated and we are not sure if it modifies the global memory or not.

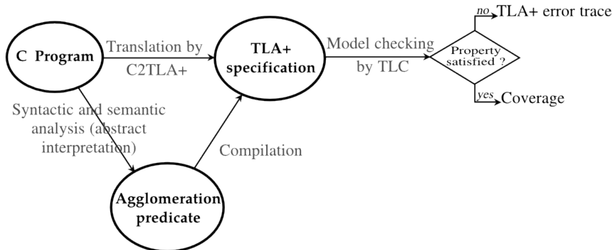


Figure 4: *Reduction process*

In this Section, we introduce the agglomeration technique by an example. Then, we describe the implementation of a safe agglomeration predicate by using a syntactic and semantic analysis on C programs. Then, we show the interest of using an unsafe agglomeration predicate to generate an abstract TLA+ specification of a C program.

In what follows, we use the expression agglomerating TLA+ actions, to designate agglomerating the corresponding statements in the C program.

4.1. An introducing example

As the semantics of a TLA+ is expressed through a state transition system, where transitions between states are ensured by TLA+ actions, the reduction technique consists in *agglomerating* consecutive actions into one atomic action which performs the effects of the original ones. The reduction idea based on agglomeration has been widely used in Petri Nets (Haddad and Pradat-Peyre 2006; Berthelot 1986).

Figure 5 shows three consecutive states linked by two actions $x' = x + 1$ and $x' = x + 2$. The result of this agglomeration (represented by $\longrightarrow$), is two states linked by one atomic action which is the result of executing the action $x' = x + 1$, then $x' = x + 2$.

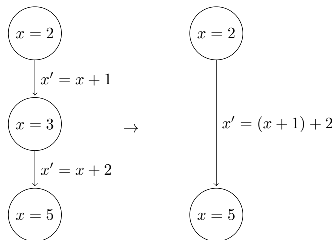
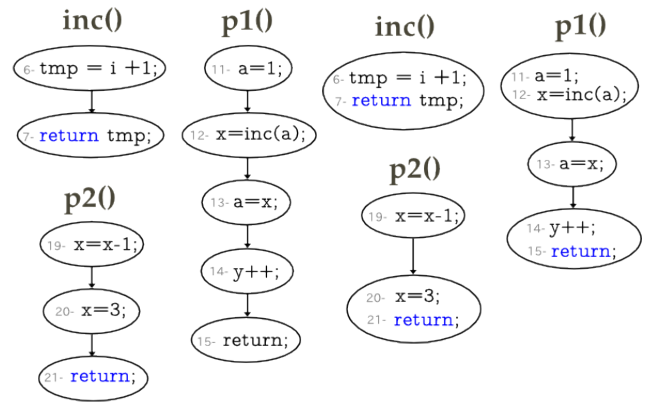


Figure 5: *Agglomerating actions*



(a) *Before reduction* ***(b)*** *After reduction*

Figure 6: *The control flow graph of a C code example*

4.2. Using syntactic and semantic analysis

Syntactic analysis

A syntactic analysis is performed to detect statements on which reduction can be applied. Often, C functions make use of local variables, and when a statement refers only to local variables, the value for which the statement is executed by a process cannot change the execution of other processes. Furthermore, we assume that statements involving local pointer variable cannot be agglomerated as they may reference shared memory.

Moreover, we consider that jump statements, namely `goto`, `break` and `continue`, can be agglomerated with its successor (designated by computing the CFG), as they only change the local register of the process ($register[id]$ in TLA+ specification).

Semantic analysis

In many scenarios, a concurrent C program could contain, in its global memory, data blocks that are accessed only by one process at a time. In that case, syntactic analysis is insufficient. Therefore, we use a semantic approximation predicate. The C program is thus analyzed and an approximation of memory access is computed using the Mthread Frama-C plugin (Mth). This latter provides information about the memory zones that are accessed concurrently by more than one process and those that are not. In this case, the agglomeration predicate is *safe* as the analysis is based on an over-approximation of the memory.

We consider the example given by Figure 2a. To illustrate the agglomeration technique on this example, we represent the C program by its control flow graph, illustrated by Figure 6a, where each state of the graph represents a C statement and edges represent the control flow. After applying the syntactic and semantic analysis on this example,

the control flow graph is transformed into a smaller graph, given in Figure 6b. Each state of the graph corresponds to one statement or a block of statements. As the `inc()` function uses only local memory, its block definition is combined into one basic block. For `p1()` function, statement on line 11 is agglomerated with statement on line 12 and statement on line 14 is agglomerated with the `return` instruction.

4.3. Generating an abstract specification

Agglomerating statements can also be useful to generate an abstract TLA+ specification of a C program. The user can define which C statements can be agglomerated using an unsafe predicate. The resulting TLA+ specification can be viewed as an abstract formal specification of the C program.

As TLA+ is a fragment of LTL$_{\setminus x}$ (Linear Temporal Logic without the "next operator"), it is well known that the equivalence between checking a property given in LTL$_{\setminus x}$ on an abstract model and checking it on the original model is ensured by the preservation (Goltz et al. 1992).

Example: Agglomerating critical sections
Consider the following fragment of C code (Figure 7) implementing an example of the producer/consumer model. The two processes share a buffer protected by a mutex `m`. The synchronization between processes is ensured by two semaphores `empty` and `full`. Mutex and semaphores are implemented as an integer values and are only accessible through two atomic operations `P()` and `V()`.

```
1  #define BUFFER_SIZE 5
2  mutex m;
3  sem full = 0, empty = BUFFER_SIZE;
4  int buffer[BUFFER_SIZE]; /* the buffer */
5  int count; /* buffer count */
6
7  void Producer(int item) {
8   while(TRUE) {
9    item = rand();/*generate a random number*/
10   P(&empty); /*acquire the empty lock*/
11   P(&m); /*acquire the mutex lock*/
12   if(count < BUFFER_SIZE) {
13     buffer[count] = item;
14     count++; }
15   V(&m); /*release the mutex lock*/
16   V(&full); /*signal full*/
17  }
18 }
19 void Consumer(void) {
20  while(TRUE) {
21   int item;
22   P(&full); /*acquire the full lock*/
23   P(&m); /*aquire the mutex lock*/
24   if(count > 0) {
25     item = buffer[(count-1)];
26     count--; }
27   V(&m); /*release the mutex lock*/
28   V(&empty); /*signal empty*/
29  }
30 }
```

Figure 7: *Example of a producer/consumer model using locks*

Translating this implementation into a TLA+ specification and model checking results in verifying all interleavings of actions between processes. We define the agglomeration predicate that states that statements protected by mutex (namely by `P()` and `V()` primitives) can be agglomerated. After reduction, the control flow graph of this example is illustrated in Figure 8b.

Therefore, the block statements from line 12 to line 14 and that from line 12 to 26 are agglomerated into one state. The state space of the TLA+ specification generated after agglomerations contains fewer states than the one without agglomerations as the reduction inside the critical section restricts the amount of interleaving allowed between processes. We define the mutual exclusion property in TLA+ as follows:

$$\begin{aligned}
&mut_exclusion(lbl1, lbl2) \triangleq \Box((\forall\, id1, id2 \in ProcSet : \\
&\ \wedge (id1 \neq id2) \wedge (Head(memory.register[id1]) \neq \langle\rangle) \\
&\ \wedge (Head(memory.register[id2]) \neq \langle\rangle) \\
&\ \wedge (Head(memory.regsiter[id1]).pc = lbl1)) \\
&\qquad \Rightarrow Head(memory.register[id2]).pc \neq lbl2)
\end{aligned}$$

This property expresses that critical sections cannot be executed simultaneously. This property was verified on the TLA+ specification after reduction. Thus, we can deduce that the property is also verified on the specification generated without the reduction technique.

4.4. Integrating the reduction into TLA+ specification

In what follows, we show how we implement the reduction on TLA+ specification. As described in Section 3, each execution of $Next$ action corresponds to executing an atomic C statement. The reduction in C programs, consists in translating a sequence of C statements into one action instead of multiple ones. Let i be the identifier of a statement and j be the identifier of its successor. To do that, we generate for each statement i a new function that we call $stmt_long_i()$ defined below.

$$\begin{aligned}
stmt_long_i(id, mem) \ &\triangleq \\
&\quad stmt_long_j(id, dispatch(id, mem))
\end{aligned}$$

The definition of $stmt_long_i(id, mem)$ consists in calling the function of the successor statement j, noted by $stmt_long_j()$ and passing as argument the memory state returned by $dispatch(id, mem)$.

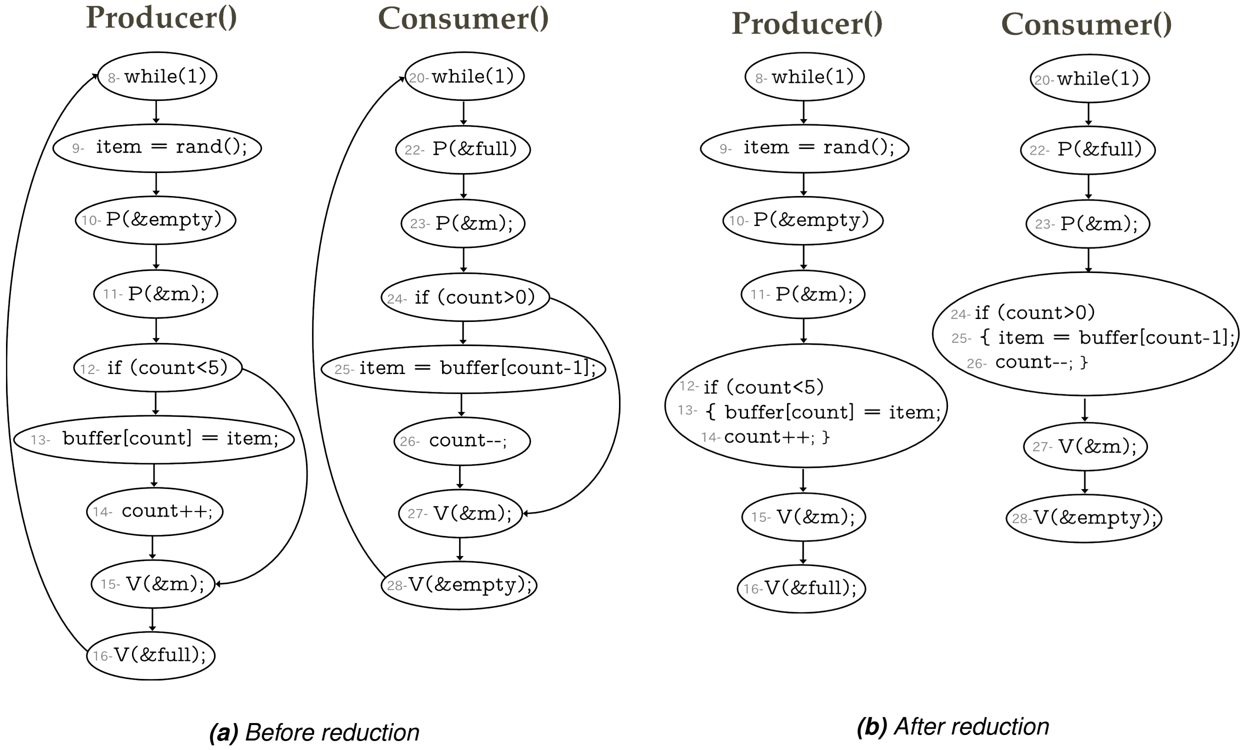


Figure 8: *The control flow graph of the `Producer()` and `Consumer()` functions before and after reduction*

To generate a reduced TLA+ specification, we iterate over all statements and when the agglomeration predicate returns true for a statement i, its translation consists in calling the function $stmt_long_j(id, dispatch(id, mem))$. Otherwise, we call $dispatch(id, mem)$ function.

The C program is thus specified by the formula $Spec$ given by equation (1), except that the $Next$ action calls a new function $dispatch_red()$, instead of $dispatch()$ function. For the example of figure 2a, the $dispatch_red()$ function is defined in Figure 9:

The $dispatch_red(id, mem)$ function calls according to the value the program counter pc contained at the head of $mem.register[id]$ the corresponding TLA+ function to execute.

5. EXPERIMENTS

The reduction technique is totally automatic and was integrated in C2TLA+ which is a Frama-C (Cuoq et al. 2012) plugin, implemented in OCaml. This Section is concerned with our practical experience. We use the Mthread plugin results and the syntactic analysis as described in Section 4 to implement our agglomeration technique.

We consider one sequential C program and four concurrent programs:

$$
\begin{array}{l}
dispatch_red(id, mem) \triangleq \\
\quad \text{CASE } Head(mem.st[id]).pc = "inc_6" \\
\qquad\qquad \rightarrow inc_long_6(id, mem) \\
\quad \Box\ Head(mem.register[id]).pc = "inc_7" \\
\qquad\qquad \rightarrow inc_long_7(id, mem) \\
\quad \Box\ Head(mem.register[id]).pc = "p1_11" \\
\qquad\qquad \rightarrow p1_long_11(id, mem) \\
\quad \Box\ Head(mem.register[id]).pc = "p1_12" \\
\qquad\qquad \rightarrow p1_long_12(id, mem) \\
\quad \Box\ Head(mem.register[id]).pc = "p1_13" \\
\qquad\qquad \rightarrow p1_long_13(id, mem) \\
\quad \Box\ Head(mem.register[id]).pc = "p1_14" \\
\qquad\qquad \rightarrow p1_long_14(id, mem) \\
\quad \Box\ Head(mem.register[id]).pc = "p2_19" \\
\qquad\qquad \rightarrow p2_long_19(id, mem) \\
\quad \Box\ Head(mem.register[id]).pc = "p2_20" \\
\qquad\qquad \rightarrow p2_long_20(id, mem) \\
\quad \Box\ Head(mem.register[id]).pc = "p2_21" \\
\qquad\qquad \rightarrow p2_long_21(id, mem) \\
\quad \Box\ \text{OTHER } \rightarrow mem
\end{array}
$$

Figure 9: *Example of the $dispatch_red()$ function definition*

Table 1: Comparing Model Checking Results with & without Reduction (time in seconds)

Prorgam	#Proc	Without reduction		With reduction		Factor
		#St	#T	#St	#T	
Zunebug	1	389	0.147	2	0.136	99.48
Dekker	2	173	0.128	70	0.109	59.53
Peterson	2	107	1.37	22	0.131	79.43
	4	1.080.161	59.2	31.221	4.82	97.10
Bakery	2	2.389	1.91	223	1.67	90.66
	4	50.515.927	1560	835.355	76.6	98.36
Philos	4	9.791.509	366	146.106	12	98.5
	5	$>$619.309.984	25340	4.179.520	352	99.32

- Zunebug which is a bug in the internal clock driver of Zune 30GB music player. The source code is taken from (Weimer et al. 2010).
- Lamport's Bakery and Peterson algorithms obtained from (Raynal 2013) and Dekker mutual exclusion algorithm presented in (Dijkstra 1968).
- Dining philosopher problem. We use the solution that appears in Tanenbaum's book (Tanenbaum 2007).

These programs make typical examples for demonstrating the strength of the state space reduction. C2LTA+ takes as input a C program and generates for each one the corresponding TLA+ specification.

Using the TLC model checker, we compute the total number of generated states and we verify a set of properties on the two specifications.

Results of experiments are shown in Table 1, where #Proc denotes the process number, #St denotes the numbers of states and #T denotes the time for model checking in seconds. Columns 3 to 6 give information about the state space generated with and without applying the reduction technique. The last column indicates the reduction factor, the ratio between the state space generated without reduction and the one after applying the reduction technique.

All experimental results were performed on an Intel Core Pentium i7-2760QM machine with 8 cores (2.40GHz each), with 8Gb of RAM memory. For zunebug, one property to verify is program termination, which is a liveness property that we express as follows:

$$termination \triangleq \diamond(Head(memory.register[1]) = \langle\rangle)$$

This property asserts that the register of the program will eventually be empty. For the TLA+ specification without agglomeration, checking this property causes TLC to report an error. This error occurs when the code takes as input the last day of a leap year, causing the code to enter into an infinite loop. After applying the reduction technique for the zunebug program, the state space size of its corresponding TLA+ specification equals 2. This is due to the fact that the program is sequential. Model checking the TLA+ code with the last day of a leap year causes the TLC model to report an incorrect recursive function definition.

For the concurrent programs, the mutual exclusion property has been successfully verified on Peterson, Bakery, Dekker and Philosophers benchmarks. As expected, the size of the state space with agglomerations is always smaller than the one without agglomerations. For the philosopher example with 5 processes, the state space without agglomeration takes more than 7 hours to be model checked. However, using the reduction technique the specification is verified in 6 minutes. The reduction factor in this cas reaches 99.32. The reduction technique obtains good results on these benchmarks due to the elimination of some intermediate states.

6. RELATED WORK

There are a wealth of research contributions on formal verification of software as well as techniques for the reduction of the state space.

Program slicing is a technique introduced by (Weiser 1981) for simplifying sequential programs for debugging and program understanding. It consists in removing from the program features that are irrelevant for the property to be verified. Recently, slicing technique has been used to reduce the state space of a system in model checking. It has been applied to Promela (Millett and Teitelbaum 2000), the input language for the Spin model checker

(Holzmann 1997). The interested reader can refer to (Tip 1995) for a detailed description of the different approaches used in the program slicing.

Predicate abstraction (Graf and Saïdi 1997) is a technique in which a set of predicates over the programs variables is used to construct an abstract program. This technique is being used in SLAM (Ball and Rajamani 2002), BLAST (Henzinger et al. 2003) and MAGIC (Chaki et al. 2004).

Other approaches perform reduction during exploration of the state space of the program. For example, partial order reduction (Valmari 1989) is a technique which explores only a representative subset of the state space of a model. The basic idea is to exploit the commutativity caused by the interleavings of transitions, which result in the same state. This technique was first introduced for checking the absence of deadlock. Subsequently, a number of variants of this technique have been developed and integrated in verification tools, like Spin (Holzmann 1997) and Verisoft (Godefroid 1997).

Although we have mentioned some projects in the C context, there are also significant works interested in model checking the Java language. For example, JPF (Visser et al. 2003) uses state compression technique to handle big states, partial order and symmetry reduction, slicing, abstraction and runtime analysis techniques to reduce the state space.

In this work, the state space reduction technique that we propose is closer to that originally introduced by (Berthelot 1986) in Petri nets formalism. Berthelot developed a large set of reduction rules for reducing the complexity of verification. Extended work has been proposed by (Haddad and Pradat-Peyre 2006). Our work differs from this latter by the fact that the model of our TLA+ specification is a state transition system and the agglomeration predicate depends on the analysis of the C program. Our reduction technique is applied during the generation of TLA+ code unlike the partial order reduction technique which performs reduction during the construction of the state space. Besides, we use TLA+ as formal framework which provides an expressive power to specify the semantics of a programming language and can reason about concurrent systems and can express safety and liveness properties unlike SLAM and BLAST which have limited support for concurrent properties as they only check safety properties.

7. CONCLUSION AND FUTURE WORK

We have proposed a technique to reduce the state space for model checking C programs. We used C2TLA+ to translate the semantics of C to the formal specification language TLA+. This reduction technique is based on an analysis phase, which defines an approximate *agglomeration predicate* that states whether a statement can be agglomerated or not. We implemented this predicate by applying a syntactic and semantic analysis on C Programs. We illustrated the effectiveness of applying the agglomeration technique to reduce the state space during the verification of C programs and also as well as to define an abstract TLA+ specification that model the behavior of C programs.

We aim to integrate a mechanism for structuring large TLA+ specifications from C programs using a refinement process between different levels of abstraction. Finally, we are planning to apply the methodology on a critical part of the microkernel of the PharOS (Lemerre et al. 2011) real-time operating system (RTOS).

REFERENCES

Mthread plugin. URL http://frama-c.com/mthread.html.

Thomas Ball and Sriram K. Rajamani. The SLAM project: Debugging System Software via Static Analysis. *SIGPLAN Not*, 2002.

Gérard Berthelot. Checking properties of nets using transformation. In *Advances in Petri Nets 1985, Covers the 6th European Workshop on Applications and Theory in Petri Nets-selected Papers*, pages 19–40, London, UK, UK, 1986. Springer-Verlag.

Sagar Chaki, Edmund M. Clarke, Alex Groce, Somesh Jha, and Helmut Veith. Modular verification of software components in c. *IEEE Trans. Software Eng.*, 30(6):388–402, 2004.

Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994.

Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C: A Software Analysis Perspective. In *Proceedings of the 10th international conference on Software Engineering and Formal Methods*, SEFM'12, pages 233–247, Berlin, Heidelberg, 2012. Springer-Verlag. URL http://frama-c.com/.

Edsger W. Dijkstra. Cooperating sequential processes. In F. Genuys, editor, *Programming Languages: NATO Advanced Study Institute*, pages 43–112. Academic Press, 1968.

Jean-Claude Fernandez, Laurent Mounier, Claude Jard, and Thierry Jron. On-the-fly verification of finite transition systems. *Formal Methods in System Design*, 1(2-3):251–273, 1992.

Patrice Godefroid. Model Checking for Programming Languages using VeriSoft. In *In Proceeedings of the 24th ACM Symposium on Principles of Programming Languages*, pages 174–186. ACM Press, 1997.

Ursula Goltz, Ruurd Kuiper, and Wojciech Penczek. Propositional Temporal Logics and Equivalences. In W.R. Cleaveland, editor, *CONCUR '92*, volume 630 of *Lecture Notes in Computer Science*, pages 222–236. Springer Berlin Heidelberg, 1992.

Susanne Graf and Hassen Saïdi. Construction of Abstract State Graphs with PVS. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 72–83, London, UK, UK, 1997. Springer-Verlag.

Serge Haddad and Jean-François Pradat-Peyre. New Efficient Petri Nets Reductions for Parallel Programs Verification. *Parallel Processing Letters*, 16(1):101–116, 2006.

Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software Verification with BLAST. pages 235–239. Springer, 2003.

Gerard J. Holzmann. The Model Checker SPIN. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

Leslie Lamport. The Temporal Logic of Actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers.* Addison-Wesley, 2002.

M. Lemerre, E. Ohayon, D. Chabrol, M. Jan, and M-B. Jacques. Method and Tools for Mixed-Criticality Real-Time Applications within PharOS. In *Proceedings of AMICS 2011: 1st International Workshop on Architectures and Applications for Mixed-Criticality Systems*, 2011.

Zohar Manna and Amir Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., New York, NY, USA, 1992.

Amira Methni, Matthieu Lemerre, Belgacem Ben Hedia, Serge Haddad, and Kamel Barkaoui. Specifying and Verifying Concurrent C Programs with TLA+. In *Formal Techniques for Safety-Critical Systems*, volume 476 of *Communications in Computer and Information Science*, pages 206–222. Springer, 2015.

Lynette I. Millett and Tim Teitelbaum. Issues in Slicing PROMELA and its Applications to Model Checking, Protocol Understanding, and Simulation. *International Journal on Software Tools for Technology Transfer*, 2(4):343–349, 2000.

George C. Necula, Scott Mcpeak, Shree P. Rahul, and Westley Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *International Conference on Compiler Construction*, pages 213–228, 2002.

Michel Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer, Heidelberg, 2013.

Andrew S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, Upper Saddle River, NJ, USA, 3rd edition, 2007.

Frank Tip. A Survey of Program Slicing Techniques. *Journal of Programming Languages*, 3:121–189, 1995.

Antti Valmari. Stubborn Sets for Reduced State Space Generation. In *Proceedings of the Tenth International Conference on Application and Theory of Petri Nets*, pages 1–22, 1989.

Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model Checking Programs. *Automated Software Engg.*, 10(2):203–232, April 2003.

Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic Program Repair with Evolutionary Computation. *Commun. ACM*, 53(5):109–116, May 2010.

Mark Weiser. Program Slicing. In *Proceedings of the 5th International Conference on Software Engineering*, ICSE '81, pages 439–449, Piscataway, NJ, USA, 1981. IEEE Press.