Specifying and Verifying Concurrent C Programs with TLA+

Amira Methni[1,4(✉)], Matthieu Lemerre[2], Belgacem Ben Hedia[1], Serge Haddad[3], and Kamel Barkaoui[4]

[1] Embedded Real-Time System Lab, CEA, LIST, 91191 Gif-sur-yvette, France
{amira.methni,belgacem.ben-hedia}@cea.fr
[2] Software Safety Lab, CEA, LIST, 91191 Gif-sur-yvette, France
matthieu.lemerre@cea.fr
[3] LSV, ENS Cachan, CNRS&INRIA, Paris, France
haddad@lsv.ens-cachan.fr
[4] CNAM, CEDRIC, Paris, France
barkaoui@cnam.fr

Abstract. Verifying software systems automatically from their source code rather than modelling them in a dedicated language gives more confidence in establishing their properties. Here we propose a formal specification and verification approach for concurrent C programs directly based on the semantics of C. We define a set of translation rules and implement it in a tool (C2TLA+) that automatically translates C code into a TLA+ specification. The TLC model checker can use this specification to generate a model, allowing to check the absence of runtime errors and dead code in the C program in a given configuration. In addition, we show how translated specifications interact with manually written ones to: check the C code against safety or liveness properties; provide concurrency primitives or model hardware that cannot be expressed in C; and use abstract versions of translated C functions to address the state explosion problem. All these verifications have been conducted on an industrial case study, which is a part of the microkernel of the PharOS real-time system.

1 Introduction

Most software systems like the Linux kernel or the Apache Webserver are implemented in a low level language such as C, which is one of the most used programming languages in industry. Verifying C code is challenging, in particular due to the presence of pointers and pointer arithmetic.

Moreover, C software systems are often concurrent, and traditional testing techniques are not efficient to check the correctness of the implementation. Thus, the use of formal verification techniques is essential. We address these issues in the context of formal verification of operating systems microkernels written in C code. In this paper, we focus on the model checking technique, a popular technique for the verification of correctness properties of finite-state systems. Given a set of properties expressed in a temporal logic and a model, it automatically

analyzes the state space of the model and checks whether the model satisfies the properties [6]. To apply this technique to the verification of C programs, the target modeling language should express all C features, handle concurrency, allow to state the properties that we want to verify, and its tools should scale up to large systems.

Contribution. Our main contribution is to provide a formal specification and verification approach of C concurrent programs, based on both axiomatic (e.g., pre-post conditions) and operational (executable model) specification of a C implementation. We use TLA+ [17] as a formal specification language for writing our specifications. In this approach, we translate a C code to an executable TLA+ specification using the C2TLA+ tool that we present in the paper. The generated specifications can be checked for runtime errors in the C code. We show how the specifications thus generated can be completed with manually written TLA+ specifications: to provide concurrency primitives, to model hardware that cannot be expressed in C, to check the C code against safety or liveness properties and to provide an abstract operational specification. In the latter case, the operational specification can be used in place of the C code in order to verify the whole system. Preliminary experiments hint that this could considerably lessen the state explosion problem. These examples are presented in a concrete case study, which is part of the microkernel of the real-time operating system PharOS [19].

Outline. The rest of the paper is organized as follows. We discuss related work in Sect. 2. We give an overview of TLA+ in Sect. 3. Section 4 presents the global approach and focus on the translation from C to TLA+. Section 5 presents a concrete application of the approach on the case study. Section 6 concludes and presents future research directions.

2 Related Work

There are a variety of formal verification techniques. Among them there are deductive verification techniques using theorem proving such as VCC [7]. These techniques provide a rigorous approach but usually require a lot of human effort and user expertise. Model checking is an automatic technique which requires less human effort because it is fully automated once the system and its properties are specified. But, it is restricted to finite-state systems. In what follows we focus on the model checking tools for C programs related to our work.

SLAM [2] was the first model checker for C programs to implement the *Counterexample Guided Abstraction Refinement* (CEGAR) approach [5]. This approach has been used later in the BLAST [11] toolkit. SLAM and BLAST have been used to check device drivers but they are only used for sequential C programs.

Besides CEGAR based tools, an approach consists to transform the C code into the input language of a model checker. Modex [14] can automatically extract a Promela model from a C code implementation. The Promela code generated

is then checked with the SPIN [12] model checker. Promela is a simple language that does not handle pointer and has no procedure calls. Modex handles these missing features by including embedded declarations and statements inside Promela specifications. The embedded code fragments can not be checked by the SPIN and can contain a division by zero error, or null pointer dereference. To mitigate this problem, Modex instruments additional checks using assertions. But, not all errors can be anticipated and the model checker can crash [13].

CBMC [4] is a bounded model checker for ANSI C programs that translates a program into a formula (in Static Single Assignment form) which is then fed to a SAT or SMT solver to check its satisfiability. It can be used to verify array bounds, pointer safety, exceptions and user-specified assertions. On the other hand, CBMC explores program behavior exhaustively but only up to a given depth, i. e., it is restricted to programs without deep loops [10]. PlusCal [18] is a high-level language for expressing multiprocess algorithms. A PlusCal algorithm can be automatically translated into a TLA+ specification. PlusCal-2 [1] improves Lamport's PlusCal language by adding new constructs like hierarchical processes and specifying atomicity for some part of the code. Moreover, it does not support some constructs of imperative programming like pointer-based structures and does not handle function calls. PlusCal is also an algorithm language that can be used to replace pseudo code but cannot be used in the final implementation.

In this work, we use TLA+ as formal framework which provides an expressive power to specify the semantics of a programming language. It is supported by the TLC model checker and the TLAPS [8] prover. Moreover, TLA+ is a logic that can reason about concurrent systems and can express safety and liveness properties unlike SLAM, BLAST and CBMC which have limited support for concurrent properties as they only check safety properties. Furthermore, TLA+ provides a mechanism for structuring large specifications using a refinement process between different levels of abstraction unlike Spin and CBMC.

3 An Overview of TLA+

TLA+ [17] is the specification language of the Temporal Logic of Actions (TLA). TLA is a variant of linear temporal logic introduced by Lamport [16] for specifying and reasoning about concurrent systems. The syntax of TLA is given in Fig. 1 (the symbol $\triangleq$ means *equal by definition*). Readers interested in a more detailed presentation of TLA+ can refer to Lamport's book [17].

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, $y + 2$ is a state function that assigns to state s two 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 actual state and primed variables refer to their values in

$\langle formula \rangle \triangleq \langle predicate \rangle \mid \Box[\langle action \rangle]_{\langle state\ function \rangle} \mid \neg\langle formula \rangle \mid \langle formula \rangle \wedge \langle formula \rangle \mid \Box\langle formula \rangle$

$\langle action \rangle \triangleq$ boolean valued expression containing constant symbols, variables, and primed variables

$\langle predicate \rangle \triangleq \langle formula \rangle$ with no primed variables | ENABLED $\langle action \rangle$

$\langle state\ function \rangle \triangleq$ nonboolean expression containing constant symbols and variables

Fig. 1. TLA syntax [17]

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.

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 linear temporal logic [20].

The behaviors satisfying this specification 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 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:

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

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

TLA+ has a 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). It requires a configuration file which defines the

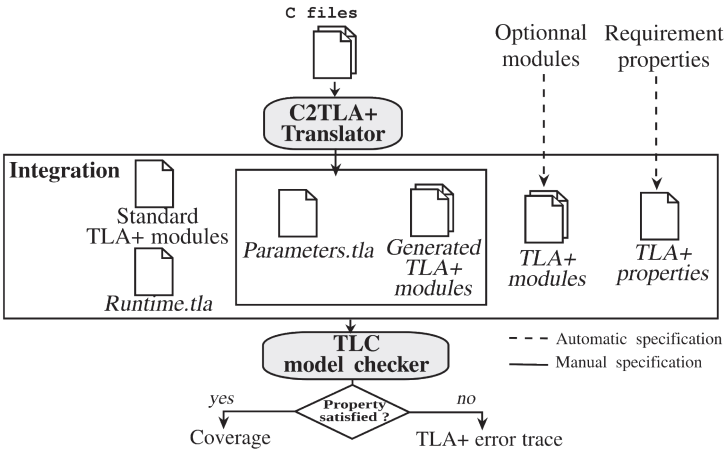


Fig. 2. Specification and verification process

finite-state instance to analyze. TLC 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.

4 Specification and Verification Process

4.1 Proposed Approach

Approach Workflow. The specification and verification process is illustrated in Fig. 2. The first step of the process is to translate from an implementation provided by one or more `.c` files a TLA+ specification using our translator C2TLA+. Before translation, the C files are parsed and normalized according to CIL (C Intermediate Language) [21]. Normalization to CIL makes programs more amenable to analysis and transformation. In particular, all expressions containing side-effects are put into separate statements (introducing temporary variables); initializers for local variables are turned into assignments; all forms of loops (`while`, `for` and `do-while`) are normalized as a single `while(1)` looping construct plus explicit `goto` statement.

After obtaining the Abstract Syntax Tree (AST) of the C program, C2TLA+ generates the TLA+ specification according to a set of translation rules described in Subsect. 4.2. The whole system is composed of TLA+ modules resulting from C translation or manual specification that come from different sources:

– Several standard modules are provided with TLA+. They contain the definition of basic operators. Like *Head*, *Tail*, *Len* (for length), ∘ (for concatenation), and *SubSeq* (for subsequence) that are defined in *Sequences* module.

- The *Runtime* module contains the TLA+ definition of arithmetic, logical and relational operators used by C2TLA+, as well as the definition of *load()* and *store()* for loading/storing an lvalue in the memory.
- Modules resulting from translation. C2TLA+ generates for each `.c` file a TLA+ module and the *Parameters* module which contains the definition of constants, type sizes, offsets of member fields and variables used by the translation. It also defines the initial predicate *Init*, the action *Next* and the specification formula *Spec*. For simplicity, we assume that the size of an integer or a pointer is 1 (one memory cell).
- Optional manual modules can be specified by the user. They provide concurrency primitives or hardware that can not be expressed in C, or an abstract model.

The set of properties is manually specified. Then, all the modules are integrated to form the complete specification, which is given to TLC to generate the model and check the properties (or refinements) to be verified. If a property is not satisfied, TLC reports a trace that leads to the bad state. TLC also provides coverage information, i. e., the number of times each action was "executed" to construct a new state. Using this information, we can identify actions that are never "executed" and which might indicate an error in the specification. Both the trace and coverage information can be translated back to C.

The Considered Subset of C. We restrict ourselves to a subset of C resulting from the simplifications done by CIL. Table 1 gives the BNF representation of the AST of CIL for this subset. The considered aspects include basic data-types (`int`, `struct`, `enum`), integer operations, arrays, pointers, pointer arithmetic, all kinds of control flow statements, function calls and recursion. Currently, we do not handle float types, non-portable conversions between objects of different types, dynamic allocation, function calls through pointers, and assignment of structs (not needed by our case study), but the translator could be updated to handle them.

4.2 Memory Layout of Concurrent C Program

A concurrent program consists in several interleaved sequences of operations called *processes* (corresponding to threads in C). C2TLA+ attributes a unique identifier to each process, and defines the constant *ProcSet* to be the set of all process identifiers.

The memory layout of a C program in C2TLA+ is organized into four regions:

- A region that contains global (and static) variables. This region is represented by a an array, called *mem*, that maps addresses to values. This memory region is shared by all processes.
- A region that contains local variables and function parameters. It is represented by the TLA+ variable *stack_data*. This region is represented by a 2-dimensional array: one dimension corresponds to the process *id* (the stack

Table 1. BNF representation of the AST of CIL for the considered subset of C (The symbols $+_{pa}/-_{pp}$ denote the addition/substraction between a pointer and an integer. $-_{pp}$ denotes the substraction between two pointers. ε is a terminal symbol that denotes an empty element).

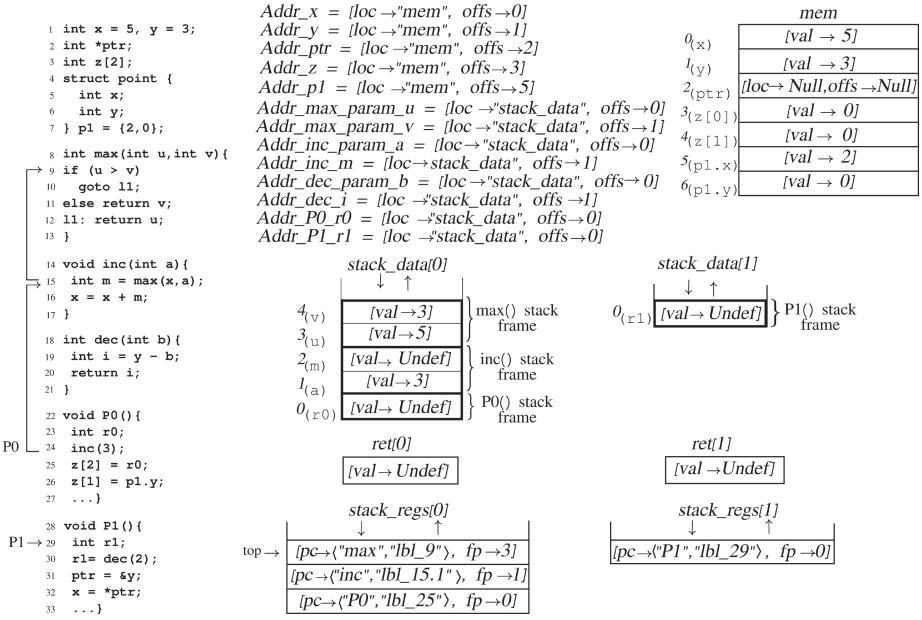
<prg>	::=<decls> (<fun_def>)*
<decls>	::= ε_{decl} | <decl> <decls>
<decl>	::=<type> VAR_ID ;
<params>	::= ε_{param} | <param> <params>
<param>	::=<type> VAR_ID ,
<fun_def>	::=<type> FUN_ID (<params>) { <decl> <stmt> }
<type>	::= int | <type> * | struct { (<type> VAR_ID ;)* };
	| enum { ENUM_ID , (ENUM_ID)* };
<stmt>	::= { (<stmt>;)* } | while(1) <stmt>
	| if <expr> <stmt> (else <stmt>)? | <lval> = <expr>
	| <lval> = FUN_ID ((<expr>,)*) | LABEL_ID: <stmt>
	| goto LABEL_ID | break | continue | return (<expr>)?
	| ε_{stmt} * Skip instruction *\
<expr>	::= <expr> <bin_op> <expr> | <un_op> <expr>
	| <expr> $+_{pa}$ <expr> | <expr> $-_{pa}$ <expr>
	| <expr> $-_{pp}$ <expr> | & <expr> | <lval> | CONSTANT
<lval>	::= VAR_ID <offs> | (* <expr>) <offs>
<offs>	::= .FIELD_ID <offs> | [<expr>] <offs> | ε_{offs}
<bin_op>	::= * | + | - | % | / | > | >= | < | <= | == | != | && | ||
<un_op>	::= !
{VAR, FUN, ENUM, LABEL, FIELD}_ID	::= [a-zA-Z][0-9a-zA-Z_]*
CONSTANT	::= [1-9]([0-9])*

is not shared between processes); the other to addresses (i. e., offsets in the stack). The stack of each function is divided into *stack frames* whose boundaries (for each process) are given in another variable, *stack_regs*. Each stack frame corresponds to a call to a function which has not yet returned. Note that this representation allows a function to access variables in its callers (through pointers), which is frequent in C.

– A region that stores the program counter of each process; i .e., which statement is being executed. This information needs to be saved and restored on function calls and returns. Rather than saving the program counter together with the data (in the *stack_data* variable), we find it simpler to organize the registers of the program as a stack. We define the TLA+ variable *stack_regs*, associating to each process a stack of records. Each record contains two fields:
  - *pc*, the program counter, points to the current statement of the function being executed, represented by a tuple ⟨function name, label⟩;
  - *fp*, the frame pointer, contains the base offset of the current stack frame.

  Note that we do not need to store the stack pointer, which is already given using "*Len(stack_data)*". Each element of the stack of records represents the registers of a function in the callstack; in particular, "*Head(stack_regs[id])*" represents the registers of the function being currently executed by the process *id*.
– A region that contains the values returned by a process. It is modeled using an array called *ret*, indexed by the process identifier.

C2TLA+ maps each C variable to unique TLA+ constant modeled by a record composed with two fields. The first one, *loc*, determines the memory region



(a) C code (b) Memory representation

Fig. 3. Example of a C code in which one process (with *id* equals 0) executes function `P0()` and the second one executes function `P1()`. The arrows in the C code indicate which statement the process *id* is executing. The top of the *stack_regs[0]* indicates that process 0 is executing the statement with label 9 of function `max()`.

where the variable is stored (*mem* or *stack_data*). The other one, *offs*, defines the offset of the data in the memory region. Fig. 3 provides a snapshot of the memory on a C code example. The TLA+ expression $[loc \mapsto \text{"}mem\text{"}, offs \mapsto 0]$ denotes the record *Addr_x* such that *Addr_x.loc* equals "*mem*" and *Addr_x.offs* equals 0. *offs* for a local variable is relative to the start of the stack frame of the current function, while *offs* for a global variable is the absolute index in *mem*.

C2TLA+ assigns to global (and static) variables not explicitly initialized the value 0 for integers, and $[loc \mapsto Null, offs \mapsto Null]$ for pointers. For local variables, it assigns the *Undef* value. *Null* and *Undef* are TLA+ "model values", which are an unspecified values that TLC considers to be unequal to any value that can be expressed in TLA+.

Loading and Assignment. An lvalue is a kind of expression that is evaluated to an address and which refers to a region of storage. Accessing the value stored in this region is performed using the *load()* operator (defined in Fig. 4) which uses the TLA+ construct IF/THEN/ELSE.

The left-hand operand of an assignment must be an lvalue. The assignment in C2TLA+ is performed by the *store()* operator defined in Fig. 5, which assigns to the lvalue *ptr* the *value* of the right-hand operand of the assignment.

$$load(id,\ ptr) \triangleq \text{IF } ptr.loc = \text{“mem”} \text{ THEN } mem[ptr.offs] \text{ ELSE } stack\_data[id][Head(stack\_regs[id]).fp + ptr.offs]$$

Fig. 4. Definition of *load()* operator

The expression $[mem \text{ EXCEPT } ![ptr.offs] = value]$ denotes the function that is equal to *mem* except that it maps the value of *ptr.offs* to *value*.

$$\begin{array}{l} store(id,\ ptr,\ value) \triangleq \\ \lor \ \land\ ptr.loc = \text{“mem”} \\ \quad \land\ mem' = [mem \text{ EXCEPT } ![ptr.offs] = value] \\ \quad \land \text{ UNCHANGED } stack\_data \\ \lor \ \land\ ptr.loc = \text{“stack\_data”} \\ \quad \land\ stack\_data' = [stack\_data \text{ EXCEPT } ![id][Head(stack\_regs[id]).fp + ptr.offs] = value] \\ \quad \land \text{ UNCHANGED } mem \end{array}$$

Fig. 5. Definition of *store()* operator

The position of a parameter or local variable in *stack_data[id]* is relative to the base of the stack frame of the current function, which equals to *Head (stack_regs[id]).fp*.

Arrays, Pointer Arithmetic and Structure Member. Accessing an array element in C2TLA+ requires computing the offset using the size of the elements, the index and the base address of the array. For example, accessing to `z[a]` is translated into:

$$load(id, [loc \mapsto Addr\_z.loc, offs \mapsto (Addr\_z.offs + (load(id, Addr\_a) * Size\_of\_int))])$$

The same kind of computation is used to perform pointer arithmetic. Similarly, accessing a structure member is achieved by shifting the base address of the structure with the constant accumulated size of all previous members. For example, accessing to `point.y` is translated into:

$$load(id, [loc \mapsto Addr\_point.loc, offs \mapsto (Addr\_point.offs + Offset\_point\_y)])$$

4.3 Intra-procedural Control Flow

Function Definition. Each C function definition is translated into an operator with the process identifier *id* as argument. The function body is translated into the disjunction of the translation of each statement it contains. A C statement is translated into the conjunction of actions that are done simultaneously. At a given state one and only one action is true (i. e., feasible). The translation of function `dec()` of the example is as follows:

```
dec(id) ≜ ∨ ∧ Head(stack_regs[id]).pc = ⟨ "dec", "lbl_19" ⟩
            ∧ store(id, Addr_dec_i, minus(load(id, Addr_y), load(id, Addr_dec_param_b)))
            ∧ stack_regs' = [stack_regs EXCEPT ![id] =
                ⟨[pc ↦ ⟨ "dec", "lbl_20" ⟩, fp ↦ Head(stack_regs[id]).fp]⟩ ∘ Tail(stack_regs[id])]
            ∧ UNCHANGED ⟨ret⟩
          ∨ ∧ Head(stack_regs[id]).pc = ⟨ "dec", "lbl_20" ⟩
            ∧ stack_regs' = ...
```

The translation of each statement s simultaneously asserts that the program counter points to s; performs the action corresponding to that statement; and updates the program counter to point to the next statement to execute.

Jump Statements. The translation of `goto`/`break`/`continue` statements consists in updating *stack_regs[id]* to the successor statement. The `goto l1` statement in function `max()` is translated as:

```
∨ ∧ Head(stack_regs[id]).pc = ⟨ "max", "lbl_10" ⟩
  ∧ stack_regs' = [stack_regs EXCEPT ![id] =
      ⟨[pc ↦ ⟨ "max", "lbl_12" ⟩, fp ↦ Head(stack_regs[id]).fp]⟩ ∘ Tail(stack_regs[id])]
  ∧ UNCHANGED ⟨mem, stack_data, ret⟩
```

Selection Statements. C integer expressions used in `if` condition are normalized by C2TLA+. Selection statement causes the program control (i. e., *stack_regs[id]*) to be transferred to a specific block based upon whether the guard expression is true or not. The translation of `if` statement in function `max()` is as follows:

```
∨ ∧ Head(stack_regs[id]).pc = ⟨ "max", "lbl_9" ⟩
  ∧ IF ((Gt(load(id, Addr_max_param_u)), (load(id, Addr_max_param_v))) ≠ [val ↦ 0])
    THEN stack_regs' = [stack_regs EXCEPT ![id] =
      ⟨[pc ↦ ⟨ "max", "lbl_10" ⟩, fp ↦ Head(stack_regs[id]).fp]⟩ ∘ Tail(stack_regs[id])]
    ELSE stack_regs' = [stack_regs EXCEPT ![id] =
      ⟨[pc ↦ ⟨ "max", "lbl_11" ⟩, fp ↦ Head(stack_regs[id]).fp]⟩ ∘ Tail(stack_regs[id])]
  ∧ UNCHANGED ⟨mem, stack_data, ret⟩
```

Iteration Statement. All loops in C are normalized by CIL as a single `while(1)` looping construct (plus eventual `if` and `break` statements), that we translate like other jump statements.

4.4 Inter-procedural Control Flow

Function Call. The function call is translated in two actions. Before calling a function f, its stack frame is pushed onto the *stack_data[id]* which obeys the LIFO order. The *stack_regs[id]* is updated by changing its head to a record whose *pc* field points to the action done once the call has finished. At the top of *stack_regs[id]* is pushed a record with *pc* pointing to the first statement of the called function, and *fp* to the new stack frame. Once the function returns, the second action copies the return value. For instance, the translation of `r1 = dec(2)` is as follows:

$$
\begin{array}{l}
\lor \land \mathit{Head}(\ \mathit{stack\_regs}[id]).pc = \langle \text{“P1”},\ \text{“lbl\_30”} \rangle \\
\quad \land \mathit{stack\_data}' = [\mathit{stack\_data}\ \textsc{except}\ !\,[id] = \mathit{stack\_data}[id] \circ \langle [\mathit{val} \mapsto 2],\ [\mathit{val} \mapsto \mathit{Undef}] \rangle] \\
\quad \land \mathit{stack\_regs}' = [\mathit{stack\_regs}\ \textsc{except}\ !\,[id] \\
\qquad = \langle [pc \mapsto \langle \text{“dec”}, \text{“lbl\_19”} \rangle, \mathit{fp} \mapsto \mathit{Len}(\mathit{stack\_data}[id]) + 1]\ \rangle \\
\qquad\quad \circ \langle [pc \mapsto \langle \text{“P1”},\ \text{“lbl\_30.1”} \rangle,\ \mathit{fp} \mapsto \mathit{Head}(\mathit{stack\_regs}[id]).\mathit{fp}] \rangle \circ \mathit{Tail}(\mathit{stack\_regs}[id])] \\
\quad \land \textsc{unchanged}\ \langle \mathit{mem},\ \mathit{ret} \rangle \\
\lor \land \mathit{Head}(\ \mathit{stack\_regs}[id]).pc = \langle \text{“P1”},\ \text{“lbl\_30.1”} \rangle \\
\quad \land \mathit{store}(id,\ \mathit{Addr\_P1\_r1},\ \mathit{ret}[id]) \\
\quad \land \mathit{stack\_regs}' = [\mathit{stack\_regs}\ \textsc{except}\ !\,[id] = \\
\qquad \langle [pc \mapsto \langle \text{“P1”},\ \text{“lbl\_31”} \rangle,\ \mathit{fp} \mapsto \mathit{Head}(\mathit{stack\_reg}[id]).\mathit{fp}] \rangle \circ \mathit{Tail}(\mathit{stack\_regs}[id])] \\
\quad \land \textsc{unchanged}\ \langle \mathit{ret} \rangle
\end{array}
$$

Return Statement. Once the function returns, the top of the *stack_regs[id]* is popped and its stack frame is removed from *stack_data[id]* using the *SubSeq* operator. The returned value is stored on *ret[id]*. The `return i` statement of function `dec()` is translated as follows:

$$
\begin{array}{l}
\lor \land \mathit{Head}(\mathit{stack\_regs}[id]).pc\ = \langle \text{“dec”},\ \text{“lbl\_20”} \rangle \\
\quad \land \mathit{stack\_regs}' = [\mathit{stack\_regs}\ \textsc{except}\ !\,[id] = \mathit{Tail}(\mathit{stack\_regs}[id])] \\
\quad \land \mathit{stack\_data}' = [\mathit{stack\_data}\ \textsc{except}\ !\,[id] = \\
\qquad \mathit{SubSeq}(\mathit{stack\_data}[id], 1, \mathit{Head}(\mathit{stack\_regs}[id]).\mathit{fp} - 1)] \\
\quad \land \mathit{ret}' = [\mathit{ret}\ \textsc{except}\ !\,[id] = \mathit{load}(id,\ \mathit{Addr\_dec\_i})] \\
\quad \land \textsc{unchanged}\ \langle \mathit{mem} \rangle
\end{array}
$$

4.5 Generating the Specification

In addition to generating constants and variables declarations, C2TLA+ also defines in *Parameters* module the main specification by generating:

- The *Init* predicate that initializes all variables of the system.
- The tuple of all variables $\mathit{vars} \triangleq \langle \mathit{mem}, \mathit{stack\_data}, \mathit{stack\_regs}, \mathit{ret} \rangle$.
- $\mathit{process}(id)$, that defines the next-state action of process id. It asserts that one of the functions is being executed until *stack_regs[id]* becomes empty. For the C code example, it is defined as:

$$
\begin{array}{l}
\mathit{process}(id) \triangleq \land\ \mathit{stack\_regs}[id] \neq \langle\rangle \\
\qquad\qquad\qquad\quad \land(\ \mathit{max}(id) \lor \mathit{inc}(id) \lor \mathit{dec}(id) \lor \mathit{P0}(id) \lor \mathit{P1}(id)\ )
\end{array}
$$

- The next-state action *Next* of all processes, that states that one of the process that has not finished is nondeterministically chosen to execute one step.

$$
\begin{array}{l}
\mathit{Next} \triangleq \lor \exists\ id \in \mathit{ProcSet} : \mathit{process}(id) \\
\qquad\qquad \lor(\forall\ id \in \mathit{ProcSet} : (\mathit{stack\_regs}[id] = \langle\rangle) \land\ (\textsc{unchanged}\ \mathit{vars}))
\end{array}
$$

- The complete specification $\mathit{Spec} \triangleq \mathit{Init} \land \Box[\mathit{Next}]_{\mathit{vars}} \land \mathit{WF}_{\mathit{vars}}(\mathit{Next})$. It is necessary to consider the fairness assumptions if we want to check liveness properties. We assume only weak fairness assumptions.

The specification can be checked by TLC without manually defining anything by the user. Errors that occur because TLC could not evaluate an expression correspond to a runtime error in the C code, like dereferencing a null pointer,

and are reported to the user. C2TLA+ also generates the *Termination* property which asserts that all processes have their stack pointer eventually empty. This property is useful in some test cases.

$$Termination \triangleq \Diamond(\forall\ id \in ProcSet\ :\ Head(stack\_regs[id]).pc = \langle\rangle)$$

5 Implementation and Experiments

C2TLA+ is developed as a Frama-C [9] plugin, implemented in OCaml. Frama-C uses CIL to reorganize and simplify C code, produces an Abstract Syntax Tree (AST) and passes it to the C2TLA+ translator. We have used C2TLA+ in a case study, described in Sect. 5.1. We use this case study as an example to describe the interactions between generated specifications and manually specified ones.

5.1 Case Study Description

We have applied our approach and tools (C2TLA+, TLC) on a critical part of the microkernel of the PharOS [19] real-time operating system (RTOS). This part contains approximately 600 lines of code and consists in a distributed version of the scheduling algorithm of the RTOS tasks. It implements a variant of the EDF (Earliest-Deadline First) scheduling algorithm. It runs on a dual-core system and consists of two processes: one running on the *control core* and the other on the *executing core*. The two processes share a set of task lists. Concurrent access to shared data is ensured by lock-free synchronization. Figure 6(a) presents the architecture of the modules of the microkernel that are of interest to us:

date provides the current date of the system. The considered implementation uses Lamport's algorithm of concurrent reading and writing of clocks [15]. This allows to read a concrete clock value, even if this value is concurrently updated.

spinlock implements lock-based concurrency primitive using "compare-and-swap" primitive.

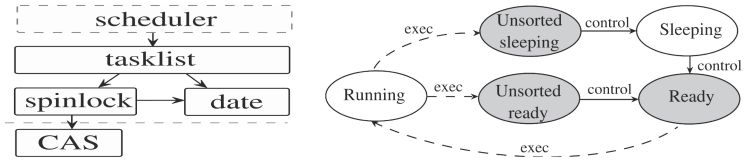
tasklist implements the life-cycle of a task as given in Fig. 6(b). Tasks can be in several states, each state corresponds to a data structure listing the tasks in that state. The incoming/outgoing edge denotes insertion/removal operation. Tasks are characterized by their *start time* and *deadline*.

scheduler is at the top-level. It performs inter-core notifications to awake processes when they have things to do. This module is not considered in translation because we do not provide support for interruptions yet.

5.2 TLA+ Modules of the Model

C2TLA+ takes as inputs the C source code of these modules. By applying our approach, we obtain the TLA+ modules of Fig. 7.

C2TLA+ generates the *Parameters* module and a TLA+ module for each C input file. These modules can interact with manually specified TLA+ modules.



(a) Modules dependencies (b) State-transition diagram of tasks

Fig. 6. Case study description

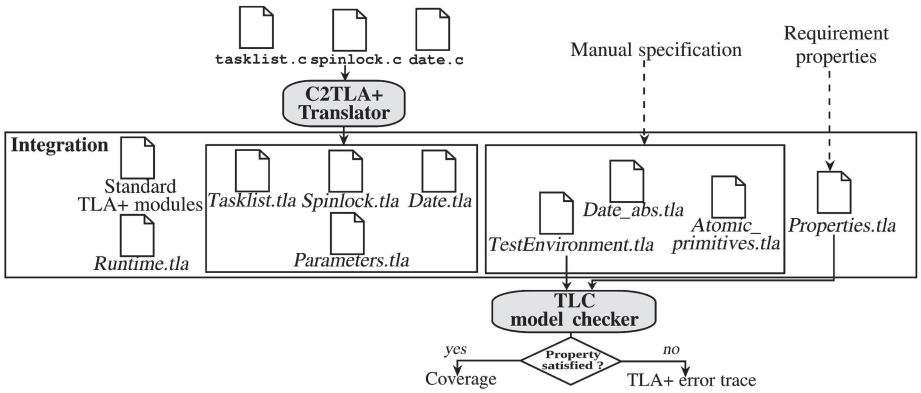


Fig. 7. TLA+ modules of the case study

Test Environment. The test environment represents the entry point of the model. It simulates the main *scheduler* module by calling the *tasklist* API and it is manually specified in the *TestEnvironment* TLA+ module.

Interacting with Manually TLA+ Specifications.

Specifying Concurrency Primitives. The *spinlock* module contains the definition of "acquire" and "release" operations which use the "compare-and-swap" (CAS) primitive. Fig. 8(a) shows the pseudo code version of this primitive. As this operation is performed atomically, we cannot translate it with C2TLA+. Such primitives are specified manually, respecting the calling conventions of Subsect. 4.4 and are declared in the C code using `__attribute__` annotation mechanism to define the TLA+ module where the primitives are specified. For instance, CAS is specified in the *Atomic_primitives* module as shown in Fig. 8(b). Other primitives could be added to *Atomic_primitives* which could be provided as a standard module.

Using an Abstract Model. The implementation of read and write operations on clock, in *date* module, is performed on several instructions. The possible interleaving of these instructions multiplies the number of states of the model. To cope with this problem, we write an abstract TLA+ version of *date*, called

```
int CAS (int *,int , int) __attribute__
((Atomic_primitives,alias("CAS")));
...
int CAS(int *addr, int old, int new)
{
 atomic {
  int temp = *addr;
  if (temp == old)
   {
     *addr = new;
     return 0;
   }
  else return 1;}
}
```

(a) Pseudo code

$$
\begin{array}{l}
CAS(id) \triangleq \\
\quad \land Head(stack\_regs[id]).pc = \langle \text{“CAS”}, \text{“lbl\_1”} \rangle \\
\quad \land \text{IF } (load(id, load(id, Addr\_CAS\_param\_addr)) = \\
\qquad load(id, Addr\_CAS\_param\_old)) \\
\qquad \text{THEN } \land mem' = [mem \text{ EXCEPT} \\
\qquad\quad ![load(id, Addr\_CAS\_param\_addr).offs] = \\
\qquad\quad load(id, Addr\_CAS\_param\_new)] \\
\qquad\quad\ \land ret' = [ret \text{ EXCEPT } ![id] = [val \mapsto 1]] \\
\qquad \text{ELSE } \land ret' = [ret \text{ EXCEPT } ![id] = [val \mapsto 0]] \\
\qquad\quad\ \land \text{UNCHANGED } \langle mem \rangle \\
\quad \land stack\_regs' = [stack\_regs \text{ EXCEPT } ![id] = \\
\qquad Tail(stack\_regs[id])] \\
\quad \land stack\_data' = [stack\_data \text{ EXCEPT } ![id] = \\
SubSeq(stack\_data[id], 1, Head(stack\_regs[id]).fp - 1)]
\end{array}
$$

(b) TLA+ code

Fig. 8. CAS definition

Date_abs which reads and writes the whole date atomically. Using this version considerably decreases the state space (see Table 2). We also verify that *Date* (the translated module) is a refinement of *Date_abs*.

5.3 Specifying and Verifying Properties

We verified various properties of the system. Here we provide some examples. We have checked that all spinlocks protect the critical sections, i. e., statements of the two processes cannot be executed simultaneously.

$$
\begin{array}{l}
Mutex(sc1, sc2) \triangleq \\
\quad \Box((Head(stack\_regs[\text{“exec\_core”}]).pc = sc1) \Rightarrow (Head(stack\_regs[\text{“control\_core”}]).pc \neq sc2))
\end{array}
$$

An important invariant of the system is that the tasks in the ready list are sorted by their deadlines; this is necessary to implement the EDF algorithm. To state this invariant, we first define a recursive operator *getSeqDeadlines* which maps the C linked list to a more abstract TLA+ sequence. The property is simpler to state on this abstract sequence by defining the *IsSortedSeq*() operator.

$$
\begin{array}{l}
getSeqDeadlines[ptr \in SetAddr] \triangleq \\
\quad \text{IF } (ptr \neq [loc \mapsto Null, offs \mapsto Null]) \\
\quad \text{THEN } \langle load(id, [loc \mapsto ptr.loc, offs \mapsto (ptr.offs + Offset\_task\_deadline)])\rangle \\
\qquad \circ getSeqDeadlines[load(\text{“unsued”}, [loc \mapsto ptr.loc, offs \mapsto (ptr.offs + Offset\_task\_next)])] \\
\quad \text{ELSE } \langle\rangle \\
IsSortedSeq(S) \triangleq \\
\quad S \neq \langle\rangle \Rightarrow (\forall\, i \in 1\,..\,Len(S), j \in 1\,..\,Len(S) : (i \neq j) \land (i \leq j) \Rightarrow (S[i].val \leq S[j].val))
\end{array}
$$

The property applied on *ready* list is expressed as follows:

$$
\Box IsSortedSeq(getSeqDeadlines[load(\text{“unused”}, Addr\_readyList)])
$$

We have also checked some liveness properties, for instance, that if a thread entered its critical section, it will eventually leave it. This property can be

expressed by comparing the program counter of the process to the statement labels of the functions "spinlock_acquire" and "spinlock_release". For example, for the *executing core*, the property is expressed as:

$$\Box((Head(stack\_regs[\text{“exec\_core”}]).pc = \langle “spinlock\_acquire”, “lbl\_2” \rangle) \Rightarrow \Diamond(Head(stack\_regs[\text{“exec\_core”}]).pc = \langle “spinlock\_release”, “lbl\_15” \rangle))$$

In order to use the abstract model *Date_abs* instead of *Date*, we have to check that the *Date* model is a refinement of the *Date_abs* model. For this, we have to map states in *Date* model with those of *Date_abs* model by substituting constants and variables used in *Date_abs* with those of *Date*. The refinement is expressed in TLA+ as logical implication. Verifying this refinement is satisfying that the specification of *Date* implies this substitution.

5.4 Verification and Discussion

We integrated the modules together and we performed model checking on two complete specifications. The first specification uses the translated *Date* module and the second one uses the abstract *Date_abs* module. The experiment was performed on an Intel Core Pentium i7-2760QM with 8 cores (2.40 GHz each) machine, with 8 Gb of RAM memory. We model checked the two specifications by considering four possible values of the clock. The executing core updates the *start time* and *deadline* of the task that has run and inserts it into the *unsorted* lists. Table 2 provides the generated states and the model checking time according to the number of tasks, for the two considered specifications.

Table 2. Runtimes of model checking (time in seconds)

Tasks	Specification using			
	Date		*Date_abs*	
	State space	Time	State space	Time
1	5.986.509	227	718.084	20
2	>501.876.263	>10.800	5.450.732	64
3	-	-	45.201.603	960
4	-	-	138.679.106	2.400

For two tasks, the specification using *Date* module takes more than 3 h to be model checked. Using an abstract model significantly reduces the size of the state space and the time required for model checking.

We have successfully checked that the correctness properties (defined in Sub-Sect. 5.3) are satisfied by the model. One of the motivations for verifying this code was to check that the fine-grained locking constructs were properly used. We checked that changing the locks in the source code leads to TLC finding

that some invariants become violated. In that case, we obtain the error trace that explains how the error can happen and TLC reports that the coverage is incomplete.

6 Conclusion and Future Work

We have sketched an approach for specifying and verifying C code based on an automated translation from C to TLA+. The main advantage of our approach is the ability to make generated TLA+ specifications from a C implementation interact with more abstract, potentially already existing manually specified TLA+ specifications. We use the TLC model checker to verify a part of the implementation of an RTOS microkernel against safety and liveness properties expressed in TLA+. We also checked that a generated specification was a refinement of an abstract TLA+ specification, and showed that we could successfully use abstraction to reduce the size of the state space.

We plan to extend this work on several interesting directions. We would like to extend the translator to handle a bigger subset of C and to generate TLA+ properties from the ACSL [3] specification language used in Frama-C. We want to update the translator so that the generated TLA+ specification catches all C runtime errors. It would be interesting to benefit from Frama-C analysis of shared variables by several processes to generate TLA+ code with less interleaving between the processes, to reduce the state space. We also plan to further study the use of TLA+ modules with different levels of refinement. Finally, we aim to use the TLA+ proof system [8] to prove properties on an abstract specification of PharOS and prove that the specification generated by C2TLA+ is a refinement of this abstract specification.

References

1. Akhtar, S., Merz, S., Quinson, M.: A high-level language for modeling algorithms and their properties. In: Davies, J. (ed.) SBMF 2010. LNCS, vol. 6527, pp. 49–63. Springer, Heidelberg (2011)
2. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. SIGPLAN Not. **37**(1), 1–3 (2002)
3. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C Specification Language, version 1.4 (2009). http://frama-c.cea.fr/acsl.html
4. Clarke, E., Kroning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
5. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
6. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)

7. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: a practical system for verifying concurrent C. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009)
8. Cousineau, D., Doligez, D., Lamport, L., Merz, S., Ricketts, D., Vanzetto, H.: TLA$^+$ Proofs. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 147–154. Springer, Heidelberg (2012)
9. Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C: a software analysis perspective. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 233–247. Springer, Heidelberg (2012)
10. D'Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. (TCAD) **27**(7), 1165–1178 (2008)
11. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with BLAST. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 235–239. Springer, Heidelberg (2003)
12. Holzmann, G.J.: The model checker SPIN. IEEE Trans. Softw. Eng. **23**(5), 279–295 (1997)
13. Holzmann, G.J.: Trends in software verification. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 40–50. Springer, Heidelberg (2003)
14. Holzmann, G.J., Smith, M.H.: An automated verification method for distributed systems software based on model extraction. IEEE Trans. Soft. Eng. **28**, 364–377 (2002)
15. Lamport, L.: Concurrent reading and writing of clocks. ACM Trans. Comput. Syst. **8**(4), 305–310 (1990)
16. Lamport, L.: The temporal logic of actions. ACM Trans. Program. Lang. Syst. **16**(3), 872–923 (1994)
17. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, MA (2002)
18. Lamport, L.: The PlusCal algorithm language. In: Leucker, M., Morgan, C. (eds.) ICTAC 2009. LNCS, vol. 5684, pp. 36–60. Springer, Heidelberg (2009)
19. Lemerre, M., Ohayon, E., Chabrol, D., Jan, M., Jacques, M.B.: 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)
20. Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems. Springer, New York (1992)
21. Necula, G.C., Mcpeak, S., Rahul, S.P., Weimer, W.: CIL: intermediate language and tools for analysis and transformation of C programs. In: International Conference on Compiler Construction. pp. 213–228 (2002)