An Approach for Verifying Concurrent C Programs

Amira Methni, Matthieu Lemerre, Belgacem Ben Hedia, Kamel Barkaoui, Serge Haddad

An Approach for Verifying Concurrent C Programs

Amira Methni, Matthieu Lemerre, Belgacem Ben Hedia
CEA, LIST,
91191 Gif-sur-Yvette, France
first.last@cea.fr

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

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

ABSTRACT

As software system and its complexity are fast growing, software correctness becomes more and more a crucial issue. We address the problem of verifying functional properties of real-time operating system (microkernel) implemented with C. We present a work-in-progress approach for formally specifying and verifying concurrent C programs directly based on the semantics of C. The basis of this approach is to automatically translate a C code into a TLA+ specification which can be checked by the TLC model checker. We define a set of translation rules and implement it in a tool (C2TLA+) that automatically translates C code into a TLA+ specification. Generated specifications can be integrated with manually written specifications that provide primitives that cannot be expressed in C, or that provide abstract versions of the generated specifications to address the state-explosion problem.

1. INTRODUCTION

Formal software verification has become a more and more important issue for ensuring the correctness of a software system implementation. The verification of such system is challenging: system software is typically written in a low-level programming language with pointers and pointer arithmetic, and is also concurrent using synchronization mechanisms to control access to shared memory locations. We address these issues in the context of formal verification of operating systems microkernels written in C programming language. We note that we are interested to check functional properties of the microkernel and not timed properties.

In this paper we present a work-in-progress approach for formally specifying and verifying C concurrent programs using TLA+ [11] as formal framework. The proposed approach is based on the translation from C code to a TLA+ specification. We propose a translator called C2TLA+ that can automatically translate from a given C code an operational specification on which back-end model checking techniques are applied to perform verification. The generated specification can be used to find runtime errors in the C code. In addition, they can be completed with manually written specifications to check the C code against safety or liveness properties and to provide concurrency primitives or model hardware that cannot be expressed in C. The manually written specifications can also provide abstract versions of translated C code to address the state space explosion problem.

Why TLA+?.

The choice of TLA+ is motivated by several reasons. TLA+ is sufficiently expressive to specify the semantics of a programming language as well as safety and liveness properties of concurrent systems [10]. Its associated model checker, TLC, is used to validate the specifications developed and is also supported by the TLAPS prover. TLA+ provides a mechanism for structuring large specifications using different levels of abstraction and it also allows an incremental process of specification refinement. So we can focus on relevant details of the system by abstracting away the irrelevant ones.

Outline.

The rest of the paper is organized as follows. We discuss related work in Section 2. We give an overview of the formal language that we used (TLA+) in Section 3. Section 4 presents the global approach and our current work. Section 5 concludes and presents future research directions.

2. RELATED WORK

There exist a wealth of work on automated techniques for formal software verification. The seL4 [9] is the first OS kernel that is fully formally verified. The verification of seL4 has required around 25 person-years of research devoted to developing their proofs and more than 150,000 lines of proof scripts. Deductive techniques are rigorous but require labor-intensive as well as considerable skill in formal logic. We focus here on more closely related work based on the model checking technique.

SLAM [2] and BLAST [6] are both software verification tools that implement the *counter-example-guided predicate abstraction refinement* (CEGAR) approach [5]. They use an automatic technique to incrementally construct abstractions i.e. abstract models cannot be chosen by user. But, SLAM cannot deal with concurrency and BLAST cannot handle recursion.

Another approach consists to transform the C code into the input language of a model checker. Modex [8] can automatically extract a Promela model from a C code implementation. The generated Promela model can then be checked with SPIN [7] model checker. As Promela does not handle pointer and has no procedure calls, Modex handles these missing features by including embedded code inside Promela specifications. On the other hand, the embedded code fragments cannot be checked by SPIN and might contain a division by zero error or null pointer dereference, Modex instruments additional checks by using assertions. But, not all errors can be anticipated and the model checker can crash [8]. CBMC [3] is a bounded model checker for ANSI C programs. It translates a C code into a formula (in Static Single Assignment form) which is then fed to a SAT or SMT solver to check its satisfiability. It can only check safety properties. CBMC explores program behavior exhaustively but only up to a given depth, i.e. it is restricted to programs without

deep loops [4].

In this work, we propose a methodology to specify and verify C software systems using TLA+ as formal framework. With TLA+, we can express safety and liveness properties unlike SLAM, BLAST and CBMC which have limited support for concurrent properties as they only check safety properties. Our approach uses abstraction and refinement in order to structure specifications and mitigate the state explosion problem for modular reasoning which is not the case of Spin and CBMC.

3. AN OVERVIEW OF TLA+

TLA+ is a formal specification language based on the Temporal Logic of Actions (TLA) [10] for the description of reactive and distributed systems. TLA combines two logics: a logic of actions and a temporal logic. To specify a system in TLA, one describes its allowed behaviors. A *behavior* is an infinite sequence of states that represents a conceivable execution of the system. A *state* is an assignment of values to variables. A *state predicate* or a *predicate* for short is a boolean expression built from variables and constant symbols. An *action* is an expression formed from unprimed variables, primed variables and constant symbols. It represents a relation between old states and new states, where the unprimed variables refer to the old state and the primed variables refer to the new state. For example, $x = y' + 2$ is an action asserting that the value of x in the old state is two greater that the value of y in the new state.

Formulas in TLA are build from actions using boolean connectives, TLA quantification and temporal operators $\Box$ (*always*). The expression $[A]_{vars}$ where A is an action and *vars* the tuple of all system variables, is defined as $A \lor vars' = vars$. It states that either A holds between the current and the next state or the values of *vars* remain unchanged when passing to the next state.

A TLA+ specification consists on a single mathematical formula *Spec* defined by:

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

where

- *Init* is the predicate describing all legal initial states,
- *Next* is the next-state action defining all possible steps of the system,
- *Fairness* is a temporal formula that specifies fairness assumptions about the execution of actions.

The formula *Spec* is true of a behavior iff *Init* is true of the first state and every state that satisfies *Next* or a "stuttering step" that leaves all variables *vars* unchanged.

To show that a property holds for a program, we must check that the program *implements* the property ϕ, which is formalized as $Spec \Rightarrow \phi$. This formula is said to be valid iff every behavior that satisfies *Spec* also satisfies ϕ.

Moreover, TLA+ has a model checker (TLC) that allows to check if a given model satisfies a given TLA formula. TLC can handle a subclass of TLA+ specifications that we believe includes most specification that describe our systems.

4. SPECIFICATION AND VERIFICATION APPROACH

The specification and verification approach is illustrated in Figure 1. The first step of the approach is to automatically translate a C code implementation to a TLA+ specification using the translator that we developed C2TLA+.

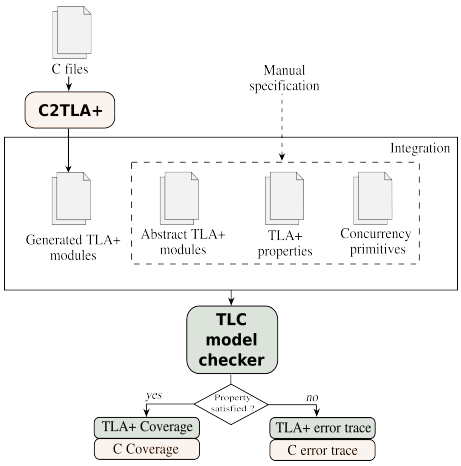

Figure 1: Specification and verification approach

C2TLA+ uses CIL [13] to transform intricate constructs of C into simpler ones. After obtaining the Abstract Syntax Tree (AST) of the normalized C code, C2TLA+ generates a TLA+ specification according to a set of translation rules that we define in Subsection 4.2. The generated specifications can be checked by TLC without any user interaction for potential C errors.

TLA+ specifications are organized into modules that can be reused independently. The methodology provides for the user the possibility to connect generated modules to other manually specified modules. These latter can model synchronization primitives like "compare-and-swap" and "test-and-set" instructions, model hardware like interruptions, or provide an abstract model of a specification. All modules are integrated together to form the whole system to verify. The user defines a set of safety and liveness properties expressed in TLA and TLC explores all reachable states in the model, looking for one in which (a) an invariant is violated, (b) deadlock occurs (there is no possible state), (c) the type declaration is violated. When a property is violated, TLC produces the minimal length trace that leads from the initial state to the bad state. To improve usability, we reimplement this trace in the C code. In addition, TLC also collects coverage information by reporting the number of times each action of a specification was "executed" to construct a new state. This information is used to generate the C code coverage of an implementation, which may be helpful for finding untested parts of the C code.

4.1 Considered features

We handle a subset of C according to simplifications done by CIL. The C aspects that we consider include basic datatypes (`int`, `struct`, `enum`), arrays, pointers, pointer arithmetic, all kinds of control flow statements, function calls, recursion and concurrency. We do not yet consider floating point operations, non-portable conversions between objects of different types, dynamic allocation, function calls through pointers, and assignment of structs. We note that these features are not needed by the system that we aim to check.

4.2 Translation from C to TLA+

4.2.1 Concurrency and memory layout

A concurrent program consists in many interleaved sequences of operations called *processes*, corresponding to

threads in C. C2TLA+ assigns to each process a unique identifier *id*.

The memory of a concurrent C program is divided by C2TLA+ into four regions:

- A region that indicates for each process where is in its program sequence. This region is modeled by a 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 $\langle$*function name*, *label*$\rangle$;
  - *fp*, the frame pointer, contains the base offset of the current stack frame.

  The top of each stack register (*Head(stack_regs[id])*) indicates the registers of the function being currently executed.
- A region that contains global (and static) variables and called *mem*. It is modeled by an array and it is shared by all processes.
- A region called *stack_data* and contains stack frames. Each process has its own stack which contains the parameters of a function and its local variables. Stack frames are pushed when calling a function and popped when returning.
- A region that contains values to be returned by processes and it is modeled by an array indexed by the process identifier, called *ret*.

Figure 2 gives an example of a C code in which one process (with *id* equals *"p1"*) executes `p1()` function and the second one (with *id* equals *"p2"*) executes `p2()` function. C2TLA+ assigns to each C variable a unique constant that we called "address". This latter specifies the memory region where data is stored (local or global) and the offset of the data in the memory region. For example, the TLA+ expression $[loc \mapsto$ "*mem*", *offs* $\mapsto 0]$ denotes the record *Addr_count* such that *Addr_count.loc* equals "*mem*" and *Addr_count.offs* equals 0.

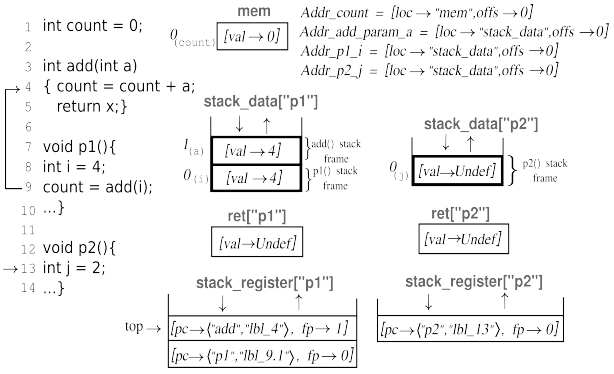

Figure 2: Example of a C code and its memory representation in TLA+

4.2.2 Loading and assignment

Variable names, fields data structure and arrays in C are lvalues. C2TLA+ translates an lvalue into an address. Loading an lvalue is performed by the TLA+ operator *load*(). An assignment of an lvalue is translated by C2TLA+ using the *store*() operator which saves the value of the right-hand operand into the memory location of the lvalue. The definition of *load*() and *store*() are given in Figure 3. For example, accessing to the value of `count` is expressed by the TLA+ expression *load(id,Addr_count)*.

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

Figure 3: Definition of *load()* and *store()* operators

4.2.3 Function definition

A C function definition is translated into an operator with the process identifier *id* as argument. Translating the function body consists of the disjunction of translating each statement that contains. C2LTA+ uses label values given by CIL to identify statements and each C statement is translated into an atomic action defined as a conjunction of sub-actions. At a given state one and only one action is evaluated to true. Each action updates the *stack_regs* variable by modifying its head by the label value of the action done once the call has finished.

4.2.4 Function call

Each function call results in a creation of a stack frame onto the local memory *stack_regs[id]* of the process *id*. The stack frame contains local variables and formal parameters which are initialized with the values with which the function was called. Then, the program counter is updated by changing its head to a record whose *pc* field points to the action done once the call has finished (the instruction following the function call). At the top of the stack register is pushed a record whose *pc* field points to the first statement of the called function, and *fp* field points to the base address of the new stack frame.

4.2.5 Return statement

Once the function returns, the returned value is stored on value returning memory *ret*. Its stack frame and the top of the stack register are popped and the program control is returned to the statement immediately following the call.

4.3 Checking the specification

C2TLA+ generates the main specification *Spec* that describes the execution of the C program.

- The *Init* predicate which specifies the initial values of all variables.
- The tuple of all variables *vars* $\triangleq$ $\langle mem, stack\_data, stack\_regs, ret\rangle$.
- The predicate *process*(*id*) defines the next-state action of the process *id*. It asserts that one of the functions is being executed until its stack register becomes empty. For the example of Figure 2, the *process*() predicate is defined as:

$$
\begin{array}{lll}
process(id) & \triangleq & \land\ stack\_regs[id] \neq \langle\rangle \\
 & & \land(\ add(id) \lor p1(id) \lor p2(id)\ )
\end{array}
$$

- The *Next* action states that one process is non-deterministically selected among those which can take an execution step or that leaves all variables unchanged when all processes terminate.

$Next \triangleq$
$\lor \exists\ id \in ProcSet : process(id)$
$\lor(\forall\ id \in ProcSet : (stack\_regs[id] = \langle\rangle) \land (\text{UNCHANGED } vars))$

- The main specification is defined by $Spec \triangleq Init \land \Box[Next]_{vars} \land WF_{vars}(Next)$. To check liveness properties in the system, we must consider fairness assumptions.

The generated specification can be directly checked by TLC. In that case, errors reported by TLC correspond to runtime errors in the C code, e.g. dereferencing null-pointer, uninitialized data access and division by zero.

4.4 Integrating abstract models

When checking whether a concurrent program satisfies a given properties, the size of the state space of the program limits the application of the model checking. A way to tackle the state space explosion problem is *abstraction*. A program usually have internal actions which need not to be considered in the verification process. Ignoring such actions reduces the state space of the program and makes the model checking feasible. With TLA+, it is possible to define different levels of abstraction and model check the existence of refinement relationship between two specifications. A specification R is a refinement of an abstract specification S iff $R \Rightarrow S$. This is true iff there exists a refinement mapping between the two specifications R and S. The refinement mapping [1] maps states of the concrete specification with states of abstract specification. From a generated TLA+ specification by C2TLA+, TLC can check if this specification refines an abstract model w.r.t. a refinement relation which preserves the properties of the abstract system.

4.5 Results and current work

Currently, we developed the translator C2TLA+ which automatically generates a TLA+ specification from C code. The translator is based on the semantics of C. We assume that generated specification behaves exactly as the C program. We tried many academic examples of C code that we checked using C2TLA+ and TLC. Actually, we are applying the methodology on a critical part of the microkernel of the PharOS [12] real-time operating system (RTOS). This part consists of a distributed version of the scheduling algorithm of the RTOS tasks. Examples of properties that we aim to check include safety properties e.g. that all spinlocks protect the critical sections, at any instant of time, the system schedules the (ready) task having earliest deadline, and also liveness properties e.g. if a thread entered its critical section, it will eventually leave it.

5. CONCLUSION AND FUTURE WORK

We have proposed an approach for specifying and verifying concurrent C programs based on an automated translation from C to TLA+. The goal of the approach is to make concrete and abstract specifications interact. Abstract models can define aspects not expressed in C code like concurrency primitives, or define an abstract specification of a concrete one. Using model checking technique, we can check the refinement relations between two specifications and the correctness properties of the whole system.

We aim to extend this work along several directions. We plan to further study the use of TLA+ modules with different levels of refinement. We also plan to check equivalence between a C code and another simplified C code. The simplified code contains less steps which would reduce the state space of the system to verify. Another avenue of future work include updating the translator to support missing features. It would be interesting to profit from data analysis in C in order to generate TLA+ code with less interleaving between the processes. Finally, we plan to use the TLA+ proof system to prove properties on an abstract TLA+ specification and prove that a generated specification by C2TLA+ is a refinement of this abstract specification.

We must remind that we are reporting the current state of a work in progress and we shall further improvement in the approach process and making it applicable on a considerable case study.

6. REFERENCES

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.

[2] T. Ball and S. K. Rajamani. The SLAM project: Debugging System Software via Static Analysis. *SIGPLAN Not*, 2002.

[3] E. Clarke, D. Kroening, and F. Lerda. A Tool for Checking ANSI-C Programs. In K. Jensen and A. Podelski, editors, *TACAS*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[4] V. D'Silva, D. Kroening, and G. Weissenbacher. A Survey of Automated Techniques for Formal Software Verification . *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[5] E. A. Emerson and A. P. Sistla, editors. *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*. Springer, 2000.

[6] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with BLAST. pages 235–239. Springer, 2003.

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

[8] G. J. Holzmann. Trends in Software Verification. In *Proceedings of the Formal Methods Europe Conference*, Lecture Notes in Computer Science, pages 40–50. Springer, 2003.

[9] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal Verification of an OS Kernel. In *SOSP*, pages 207–220, New York, USA, 2009.

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

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

[12] 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.

[13] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. CIL: Intermediate Language and Tools for Analysis and Transformation of C Programs. In *International Conference on Compiler Construction*, pages 213–228, 2002.