# Verifying and Constructing Abstract TLA Specifications: Application to the Verification of C programs

Amira Methni*, Matthieu Lemerre†, Belgacem Ben Hedia†, Serge Haddad‡ and Kamel Barkaoui*
*CNAM, CEDRIC, 292 rue Saint Martin, Paris Cedex 03, France
Email: first.last@cnam.fr
†CEA, LIST, Centre de Saclay, PC172, 91191, Gif-sur-Yvette, France
Email: matthieu.lemerre@cea.fr, belgacem.ben-hedia@cea.fr
‡LSV, ENS Cachan, CNRS & INRIA, France
Email: haddad@lsv.ens-cachan.fr

*Abstract*—**One approach to verify the correctness of a system is to prove that it implements an executable (specification) model whose correctness is more obvious. Here, we define a kind of automata whose state is the product of values of multiple variables that we name State Transition System (STS). We define the semantics of TLA+ (specification language of the Temporal Logic of Actions) constructs using STSs, in particular the notions of TLA+ models, data hiding, and implication between models. We implement these concepts and prove their usefulness by applying them to the verification of C programs against abstract (TLA+ or STS) models and properties.**

***Keywords–Temporal Logic of Actions; formal specification; model-checking; C programs; refinement mapping.***

## I. Introduction

As software systems become large and error-prone, formal verification methods become an essential key concept to ensure their correctness. Model Checking [1] provides an automated technique to check and detect errors in computer programs. But despite its promise, the verification process may be complex due to the size of these systems. One useful technique to reduce the complexity of verification process is abstraction. Generally, an abstract model specify "what" the system do while the concrete model describes "how". The idea is to map the concrete set of states to a smaller set of states resulting in an approximation of the system with respect to the property of interest. We say that the concrete model implements the abstract one. Verifying the abstract model is generally more efficient than verifying properties of the original.

*a) Contributions:* We define an operational semantics of a TLA specification in terms of automata, that we called State Transition System (STS). We remind the concepts of implementation relation and refinement mapping in TLA+ that we formalize in terms of relations between STSs. The refinement between specifications can be checked with the TLC model checker. Verified properties on the abstract specification can thus be deduced in the concrete specification. A way to abstract details of the concrete specification is to hide its irrelevant variables. TLA+ can express data hiding, but TLC can't support this type of construct. So, we have implemented the notion of data hiding by constructing a STS that we call "quotient STS", which is constructed by extending the TLC model checker. In order to let the quotient STS be analyzed by existing tools, we extend the TLC model checker to produce an LTS that can be checked by the CADP toolkit. We apply the mentioned concepts on C programs using our tool C2TLA+. Preliminary results show the importance of using an abstract model to reduce the complexity of verification.

*b) Outline:* The remainder of the paper is structured as follows. We give an overview of TLA+ and its operational semantics in Section 2. Section 3 reminds the concepts of refinement mapping and the implementation relation between specifications and describe a way to construct the quotient STS. In Section 4, we apply these concepts to verify the correctness of the C implementation with respect to their specification and we report some preliminary experimental results obtained. We discuss related work in Section 5. Section 6 concludes and presents future research directions.

## II. An Operational Semantics for TLA specification

In this section, we explain some basics concerning the syntax and the semantics of TLA [2]. Then, we describe the operational semantics of TLA using a STS.

### A. Overview of TLA+

TLA+ is a formal specification language based on the TLA [3] for the description of reactive and distributed systems. TLA itself is a variant of linear-time temporal logic. The semantics of TLA is defined in terms of *states*. A *state* is a mapping from variables to values. A *state function* is a nonboolean expression built from constants, variables and constant operators, that maps each state to a value. For example, $y + 3$ is a state function from a state $s$ to three 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 the next-state. Thus, an action represents a relation between an old state and a 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. A *state predicate* (or predicate for short) is an action with no primed variables.

Syntactically, 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 ($\forall$, $\exists$), the operators ′ and the unary temporal operator $\Box$ (*always*) of linear-time temporal logic [4].

The expression $[A]_{vars}$ where $\mathcal{A}$ is an action and $vars$ the tuple of all system variables, is defined as $\mathcal{A} \vee (vars' =$

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-438-1 

$vars$). It states that either $\mathcal{A}$ holds between the current and the next state or the values of $vars$ remain unchanged when passing to the next state. For any action $\mathcal{A}$, the state predicate $Enabled(\mathcal{A})$ describes whether the action $\mathcal{A}$ can be executed in the current state $s$, i.e., there exists some state $t$ such that $s \longrightarrow t$ is an $\mathcal{A}$ step.

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. The system specification can be given by the temporal formula $\Phi$ defined as a conjunction of the form:

$$\Phi \triangleq Init \wedge \Box[Next]_{vars} \wedge F \quad (1)$$

Where, $Init$ is the predicate describing all legal initial states, $Next$ is the next-state action defining all possible transitions between states and $F$ is a conjunction of fairness assumptions about the execution of actions. However, other forms of specification are possible and can occasionally be useful.

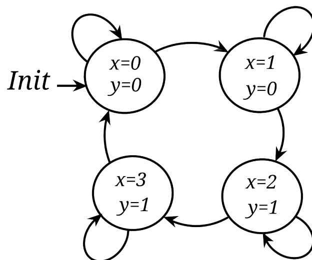
A TLA formula is true or false on a behavior, which is a sequence of states. Let $\sigma = \langle s_0, s_1, \ldots \rangle$ be a behavior. $\sigma$ satisfies $Spec$ iff $Init$ is true of the first state $s_0$ and every state that satisfies $Next$ or a "stuttering step" that leaves all variables unchanged.

### B. State Transition System

In TLA, the behavior of a system is modeled as an infinite sequence of states. The operational semantics of a TLA specification can be given in terms of a STS, which is easier to work with than sets of sequences.

$$\begin{array}{lll} \Phi & \triangleq & \wedge\ (x = 0 \wedge y = 0) \\ & & \wedge\ \Box\ [\ \begin{array}{ll} \wedge & x' = (x+1)\%4 \\ \wedge & y' = x \div 2 \end{array}\ ]_{\langle x,y \rangle} \end{array}$$

(a) TLA specification




(b) The STS of $\Phi$

Figure 1. The operational semantics of a TLA specification

*Definition 1:* A STS is a 3-tuple $\mathcal{T} = (\mathcal{Q}, \mathcal{I}, \delta)$ given by

- a finite set of states $\mathcal{Q}$,
- a set $\mathcal{I} \subseteq \mathcal{Q}$ of initial states,
- a transition relation $\delta \subseteq \mathcal{Q} \times \mathcal{Q}$.

Figure 1 shows a TLA specification and its corresponding STS $\mathcal{T}_\Phi = (\mathcal{Q}_\Phi, \mathcal{I}_\Phi, \delta_\Phi)$ which encodes all its possible behaviors ($\div$ symbol denotes integer division). The specification $\Phi$ is translated into $\mathcal{T}_\Phi$ as follows:

- $\mathcal{T}_\Phi$ has initial state(s) $\mathcal{I}_\Phi$ specified by the predicate $x = 0 \wedge y = 0$,
- every state $s \in \mathcal{Q}_\Phi$ corresponds to a valuation of the state function $\langle x, y \rangle$,
- each transition $t \in \delta_\Phi$ corresponds to satisfying the predicate $[x' = (x+1)\%4\ \wedge y' = x \div 2]_{\langle x,y \rangle}$.

## III. Refinement and Abstraction of TLA Specifications

A way to reduce the verification task is to define an abstract model as a specification, and then relate behaviors of the abstract model to those of the implementation. Properties checked on the abstract model can be deduced on the concrete one. We use concrete model to refer to high-level specification and abstract model to refer to low-level specification. This section describes the semantics of refinement between a high-level and a low-level TLA+ specification. Then, we present a way to automatically construct a reduced model, which abstracts the detailed behavior of the concrete TLA+ specification.

### A. Refinement Mapping

Abadi and Lamport [5] described that a high-level specification $\Psi$ implements a low-level specification $\Phi$ iff for each behavior of $\Psi$, there is a behavior of $\Phi$ with the same sequence of externally visible states, allowing stuttering, e .g., if the state $\Phi$ does not change during a finite number of steps. This implementation relation is proved by defining a refinement mapping between specifications.

Let $\Psi$ and $\Phi$ be two TLA specifications, $x_1, \ldots, x_m$ and $y_1, \ldots, y_n$ the variables occurring in the specifications $\Psi$ and $\Phi$ respectively. A (concrete) specification $\Psi$ implements an abstract specification $\Phi$ if $\Psi \Rightarrow \Phi$. The proof of this implication consists in defining state functions $\bar{y_1}, \ldots, \bar{y_n}$ in terms of the variables $y_1, \ldots, y_n$ and prove that $\Psi \Rightarrow \bar{\Phi}$, where $\bar{\Phi}$ denotes the formula $\Phi$ obtained by substituting $\bar{y_i}$ for the free occurrences of $y_i$, for all $i$.

The set of state functions $\bar{y_1}, \ldots, \bar{y_n}$ is called a *refinement mapping*. The "barred variable" $\bar{y_i}$ is the state function with which $\Psi$ implements the variable $y_i$ of $\Phi$. So, if $\sigma$ is the behavior $s_1 \rightarrow s_2 \rightarrow s_3 \ldots$ of $\Psi$, we define the behavior $\bar{\sigma}$ to be $\bar{s_1} \rightarrow \bar{s_2} \rightarrow \bar{s_3} \ldots$ We say that $\Psi$ implements $\Phi$ under this refinement mapping iff, for each behavior $\sigma$ satisfying $\Psi$, the behavior $\bar{\sigma}$ is a behavior of $\Phi$.

### B. Implementation Relation and Property Preservation

The proof $\Psi \Rightarrow \Phi$ under a refinement mapping is sufficient to verify that $\Psi$ implements $\Phi$ [5]. The key to the implication relation is that TLA allows to write only formula that are *insensitive to stuttering*, i.e., given a TLA formula $\Phi$ and two stuttering equivalent runs $\sigma$ and $\sigma'$, $\Phi$ holds along $\sigma$ if and only if it holds along $\sigma'$ [3]. This implementation relation between TLA specifications can be viewed as a weak simulation relation between its corresponding STSs.

*Definition 2:* Let $\mathcal{T}_\Psi = (\mathcal{Q}_\Psi, \mathcal{I}_\Psi, \delta_\Psi)$ and $\mathcal{T}_\Phi = (\mathcal{Q}_\Phi, \mathcal{I}_\Phi, \delta_\Phi)$ denote two STSs. A simulation $\mathcal{R}$ relation from $\mathcal{Q}_\Psi$ to $\mathcal{Q}_\Phi$ is a function that satisfies the following conditions:

- $\forall s \in \mathcal{I}_\Psi, \mathcal{R}(s) \subseteq \mathcal{I}_\Phi$ (initial states are mapped to initial states),

- For each state pairs $(s_1, s_2) \in \delta_\Psi$, $(\mathcal{R}(s_1), \mathcal{R}(s_2)) \in \delta_\Phi$ (state transitions are mapped into state transitions or stuttering steps).

If a lower-level specification, expressed by a TLA formula $\Psi$, implements an abstract specification $\Phi$, $\Psi$ preserves all TLA properties of $\Phi$ if and only if for every formula $\phi$, if $\Phi \Rightarrow \phi$ is valid, then so is $\Psi \Rightarrow \phi$. This is true if $\Psi \Rightarrow \Phi$.

### *C. Data Hiding in TLA*

A very useful form of data abstraction is variable hiding, which refers to providing only essential information to the outside world and hiding not needed information. In TLA, it is possible to hide some variables using the existential quantifier $\exists$ (which differs from the quantifier of predicate logic). The formula $\exists\, x : \Phi$ asserts that it doesn't matter what the actual values of $x$ are, but there are some values $x$ can assume for which $\Phi$ holds. The meaning of $\exists$ is defined by (2). The formula $\sigma \sim_x \tau$ is defined to be true iff $\sigma$ can be obtained from $\tau$ (or vice-versa) by adding and/or removing stuttering steps and changing the values of $x$. Thus, the (2) is true for a behavior $\sigma$ iff $\Phi$ is true for some behavior $\tau$ such that $\sigma \sim_x \tau$ is true.

$$\sigma \models \exists\, x : \Phi \triangleq \exists_{\ behavior} (\sigma \sim_x \tau) \wedge (\tau \models \Phi) \quad (2)$$

The temporal formula (3) describes a specification $\Phi$ where $v$ is the list of all relevant state variables and $x$ is the list of internal (hidden) variables.

$$\Phi \triangleq \exists\, x : Init \wedge [Next]_v \wedge L \quad (3)$$

The existential operator is a very simple and useful way in which the system is described as a black box. However, in practice, the TLC model checker cannot handle the TLA hiding operator. In what follows, we present a way to implement data hiding by constructing a quotient STS from a TLA specification.

### *D. Computing a Quotient STS*

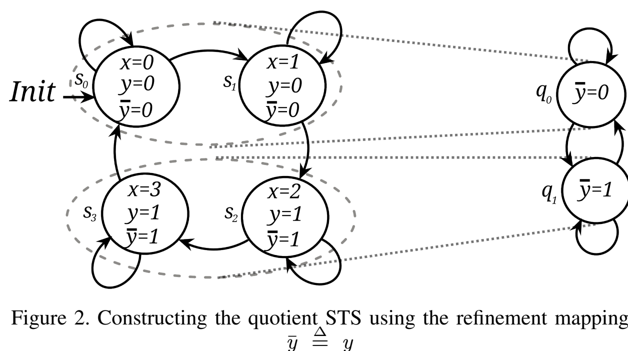



Figure 2. Constructing the quotient STS using the refinement mapping $\bar{y} \triangleq y$

Given a concrete STS $\mathcal{T} = (\mathcal{Q}, \mathcal{I}, \delta)$ describing a TLA specification, one can obtain an abstraction of $\mathcal{T}$, a small STS that we call *quotient STS* and which is obtained by quotienting the states $\mathcal{Q}$ under a refinement mapping $\gamma$.

Figure 2 shows a STS resulting from adding a refinement mapping $\bar{y} \triangleq y$ in all states of the concrete STS. The quotient STS (at the right side of the figure) is constructed by collapsing all states related under the relation $\gamma$ into the same state. Let $\mathcal{T}/_\gamma = (\mathcal{Q}/_\gamma, \mathcal{I}/_\gamma, \delta/_\gamma)$ be the quotient STS of $\mathcal{T} = (\mathcal{Q}, \mathcal{I}, \delta)$

```
1: procedure QUOTIENTSTS
2:     Q_γ ← γ(I)
3:     NotSeen ← {s′ ∈ Q | s ∈ Q and (s, s′) ∈ δ}
4:     while NotSeen ≠ {} do
5:         for ∀q ∈ NotSeen do
6:             if γ(q) ∉ Q_γ then
7:                 Q_γ ← Q/_γ ∪ {γ(q)}
8:                 δ/_γ = δ/_γ ∪ {(γ(q), γ(q′)) | (q, q′) ∈ δ}
9:                 NotSeen = NotSeen \ {q}
10:            end if
11:        end for
12:    end while
13: end procedure
```

Figure 3. Construction algorithm of the quotient STS

under the refinement mapping $\gamma$. The algorithm of constructing $\mathcal{T}/_\gamma$ is given in Figure 3.

We extend the implementation of TLC to produce the quotient STS "on-the fly" when the TLC model checker computes the state space of a specification. In fact, TLC makes efficient use of disk. It doesn't keep all states in memory which is the limiting factor of the explicit other model checkers. Instead, it stores just fingerprints of states, which is a 64-bit number generated by a "hashing" function. So, the probability that two states have the same fingerprint is $2^{-64}$ which is a very small number. So, the quotient STS is generated with the same fingerprint collision probability and without exploding the memory.

### *E. Translating a STS into a Labelled Transition System*

In order to use existing tools to check properties on a STS, we transform the quotient STS into a Labelled Transition System (LTS), that we call quotient LTS.

*Definition 3:* A LTS is 4-tuple $\mathcal{T} = \langle \mathcal{Q}, \mathcal{L}, \delta, s_0 \rangle$, where:

- $\mathcal{Q}$ is the set of states,
- $\mathcal{L}$ is the set of action labels,
- $\delta$ is the transition relation (a subset of $\mathcal{Q} \times \mathcal{L} \times \mathcal{Q}$),
- and $s_0$ is the initial state.

A transition $(s_1, l, s_2)$ of $\delta$, indicates that the system can move from state $s_1$ to state $s_2$ by performing action labelled by $l$.

*c) Property preservation:* The equivalence between checking a property given in $\text{LTL}_{\setminus x}$ (Linear Temporal Logic without the "next operator") on the quotient LTS and checking it on the original LTS is ensured by the preservation.

*Proposition 1:* Let $\varphi$ be an $\text{LTL}_{\setminus x}$ formula, let $\mathcal{T}_\Phi$ and $\mathcal{T}_\Psi$ be two STSs such that $\mathcal{T}_\Psi \Rightarrow \mathcal{T}_\Phi$. If $\mathcal{T}_\Phi \models \varphi$ then $\mathcal{T}_\Psi \models \varphi$.

### *F. Usefulness of the Quotient LTS*

The quotient LTS abstracts away the details of the concrete specification. Its main advantage is its small size. As properties are preserved between the concrete specification and its corresponding quotient STS, model checking properties can be done on the quotient LTS directly, which is a simple task. The quotient LTS is generated once and can be used to verify different properties (modulo the refinement mapping).

To express and check properties on the quotient STS, we use the CADP [6] toolkit. For this, we first adapt the label

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-438-1

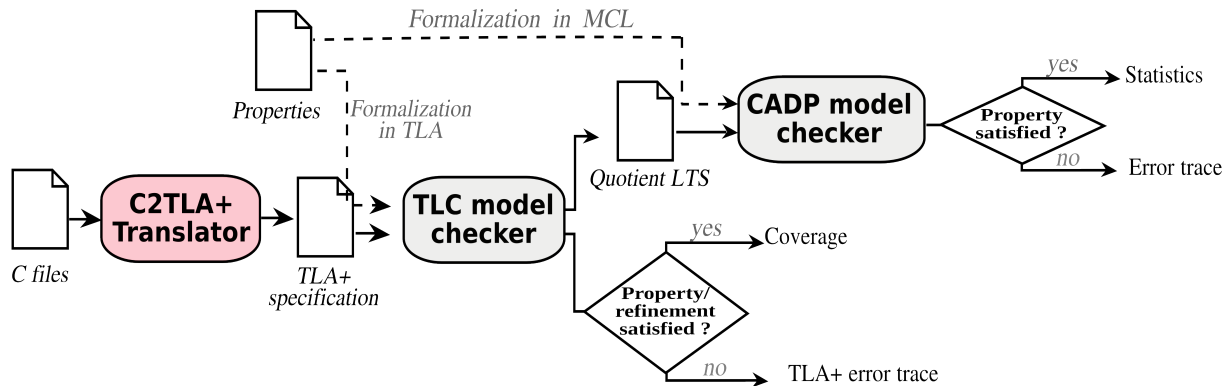



Figure 4. Verification flow of C programs

names such that LTS can be parsed by the CADP tools. Then, we express properties in the Model Checking Language (MCL) [7] language, the property specification language of CADP that can be verified by its associated model checker.

## IV. Application of C programs

In this section, we implement the concept of refinement between TLA+ specifications and the quotient LTS on C programs. Figure 4 illustrates the verification flow of C programs. We use our tool C2TLA+ [8] to translate C programs into (a concrete) TLA+ specification. This latter can be checked directly against a set of properties, or against an abstract specification by defining the refinement mapping and the implementation relation between the concrete and the abstract specifications. Properties can be expressed in TLA to be verified using the TLC model checker. The quotient LTS is generated, and MCL properties can be verified by the CADP model checker.

In what follows, we briefly present how we specify the semantics of C in TLA+. We apply the described notions by considering the example of the dining philosophers. Finally, we assess the usefulness of using abstraction by giving results of properties verification using TLC and the CADP model checker.

### A. *TLA+ specification of a C program*

C2TLA+ [8] generates a TLA+ specification that describes the behavior of the C program as a closed system according to a set of translation rules. A concurrent program consists in a set of C functions. In C2TLA+, concurrency is modeled by considering all possible interleaving of sequences of operations called *processes* (corresponding to threads in C). Each step of the complete specification is attributed to exactly one process. The C program is defined by a TLA formula in the form of (1). For more detailed information about the translation from C to TLA+, please refer to our previous work [8].

### B. *Illustrating Example*

As an example, we consider the classic dining philosophers problem. One possible solution to this problem is the one that appears in Tanenbaum's popular operating systems textbook [9],, given in Figure 5.

In the implementation of this solution, the global semaphore `mutex` provides mutual exclusion for execution

```
#define N 4
#define THINKING 0
#define HUNGRY 1
#define EATING 2
#define LEFT(i) (i+N-1)%N
#define RIGHT(i) (i+1)%N
typedef int semaphore;
int state[N];
semaphore mutex;
semaphore sem[N];

void philosopher(int i)
{ while (1) {
   think();
   take_forks(i);
   eat();
   put_forks(i); }
}
void take_forks(int i) {
  P(&mutex);
  state[i] = HUNGRY;
  test(i);
  V(&mutex);
  P(&sem[i]);}

void put_forks(i)
{
  P(&mutex);
  state[i] = THINKING;
  test(LEFT(i));
  test(RIGHT(i));
  V(&mutex);
}

void test(i)
{
  if (state[i] = HUNGRY
      && state[LEFT(i)]!=
           EATING
      && state[RIGHT(i)]
          != EATING)
   {
    state[i] = EATING;
    V(&sem[i]);
   }
}
```

Figure 5. Tanenbaum's solution for the dining philosophers

of critical sections and the semaphore `sem[i]` ensures synchronization. The latters perform `P()` to acquire a lock and `V()` to release it, using "Compare-and-swap" primitive.

### C. *Refinement of Specifications*

*d) Abstract specification of the dining philosophers:* We define a coarse-grained representation of the dining philosopher, illustrated by Figure 6 that captures the aspects of the system that interest us without giving all the details of its internal structure.

In order to check liveness properties, we consider that the philosopher cannot starve waiting for a fork, i.e., no philosopher is eating forever. This assumption is stated by the formula $\mathit{Fairness}$, where $\mathrm{WF}_{vars}(\mathcal{A})$ denotes weak fairness on action $\mathcal{A}$ and the symbol $\Diamond$ denotes the temporal operator *eventually*.

Copyright (c) IARIA, 2015. ISBN: 978-1-61208-438-1 

MODULE *Abstract_philosophers*

```
EXTENDS Naturals, TLC
CONSTANT N
VARIABLES phil_state, forks
vars ≜ ⟨phil_state, forks⟩
fork_available(i) ≜ forks[i] = N
fork_acquire(p, i) ≜ forks' = [forks EXCEPT ![p] = i]
forks_release(p) ≜
        forks' = [forks EXCEPT ![p] = N, ![(p + 1)%N] = N]
fork_release(p) ≜ forks' = [forks EXCEPT ![p] = N]
LEFT(p) ≜ (p + 1)
RIGHT(i) ≜ IF (i = 0) THEN (N − 1) ELSE (i − 1)

think(ph) ≜
    ∧ phil_state[ph] = "think"
    ∧ fork_available(LEFT(ph))
    ∧ fork_acquire((LEFT(ph), ph)
    ∧ phil_state' = [phil_state EXCEPT ![ph] = "hungry"]

hungry(ph) ≜
    ∧ phil_state[ph] = "hungry"
    ∧ IF (fork_available(ph))
      THEN
        ∧ fork_acquire(ph, ph)
        ∧ phil_state' = [phil_state EXCEPT ![ph] = "eat"]
      ELSE
        ∧ fork_release(LEFT(ph))
        ∧ phil_state' = [phil_state EXCEPT ![ph] = "think"]

eat(ph) ≜ ∧ phil_state[ph] = "eat"
          ∧ forks_release(ph)
          ∧ phil_state' = [phil_state EXCEPT ![ph] = "think"]

Init ≜ ∧ phil_state = [i ∈ (0 . . (N − 1)) ↦ "think"]
       ∧ forks = [i ∈ (0 . . (N − 1)) ↦ N]

Spec ≜ Init ∧ □[∃ i ∈ 0 . . (N − 1) :
                 think(ph) ∨ hungry(ph) ∨ eat(ph)]_vars
           ∧ Fairness
```

Figure 6. Abstract TLA+ version of the dining philosophers

```
Fairness ≜
    ∧ ∀ i ∈ (0 . . N − 1) : WF_vars(hungry(i)) ∧ WF_vars(eat(i))
    ∧ ∀ i ∈ (0 . . N − 1) : □◇(ENABLED ⟨think(i)⟩_vars)
                       ⟹ (□◇⟨eat(i)⟩_vars)
```

*e) Specifying the refinement relation:* To check that the concrete specification generated by C2TLA+, implements the abstract version of the dining philosophers, we define the refinement relation as shown in Figure 7. In this section, we don't illustrate the translation of the C code, as the translation rules are described in our previous work [8].

The implementation relation is an implication formula $Spec \Rightarrow Abstract_instance!Spec$.

### D. Expressing properties

An interesting property that the implementation should hold is that the critical sections are protected with the primitives `P()` and `V()`. This property can be simply expressed in TLA+ (on the abstract specification) as follows:

MODULE *refinement_definition*

```
EXTENDS Concrete_philosophers
philNum ≜ load("unused", Addr_N)
get_val(addr, off) ≜
    load("unused", [loc ↦ addr.loc, offs ↦ addr.offs + off]).val

refmap(addr) ≜
    [i ∈ (0 . . philNum) ↦
         LET val ≜ get_val(addr_state, i)
         IN  IF val = 0 THEN "think"
             ELSE IF val = 1 THEN "hungry"
                  ELSE "eat" ]

Abstract_instance ≜ INSTANCE Asbtract_philosophers WITH
        N ← philNum,
        phil_state ← refmap(Addr_state)

Spec ⟹ Abstract_instance!Spec
```

Figure 7. Definition of refinement relation between abstract and concrete TLA+ specifications of the dining philosophers

```
mutual_exclusion ≜
∀ i ∈ (0 . . (N − 1)) : (phil_state[i] = "eat") ⟹
(phil_state[LEFT(i)] ≠ "eat" ∧ phil_state[RIGHT(i)] ≠ "eat")
```

The dining philosophers problem captures many aspects of liveness. Among liveness properties of the dining philosophers is starvation-freedom and deadlock freedom that we expressed in TLA+ as follows:

```
NoStarvation ≜ ∀ i ∈ (0 . . (N − 1)) :
  □((phil_state[i] = "hungry") ⟹ ◇(phil_state[i] = "eat"))

DeadlockFree ≜
  □((∀ i ∈ (0 . . (N − 1)) : (phil_state[i] = "hungry")) ⟹
       (∀ i ∈ (0 . . (N − 1)) : ◇(phil_state[i] = "eat")))
```

### E. Verification results and comparison

We check that the concrete TLA+ specification (generated by C2TLA+) implements the abstract TLA+ specification (given in Figure 7). We also check the set of properties on these two specifications. We extract the quotient LTS from the concrete specification that we checked against the set of properties that we express in MCL. Table I shows the number of states and the verification time of the concrete and the abstract specifications using TLC, and the numbers of states, transitions and the time verification of the quotient LTS using CADP model checker. Experiments were carried on an Intel Core Pentium i7-2760QM with 8 cores (2.40GHz each) machine, with 8Gb of RAM memory. For 5 philosophers, the state space of the concrete TLA specification exceeds 113 millions states and its verification takes more than 10 hours to check the properties.

For the same number of philosophers, the abstract TLA specification generates 82 states and properties were checked in only 1 minute using TLC. On the other hand, the quotient LTS generated 47 states and its verification time is 42s. Due to the preservation properties, we can deduce that all the verified properties on the abstract TLA specification or on the quotient LTS are verified on the concrete specification. The use of

TABLE I. RUNTIMES OF MODEL CHECKING

| Philos | Verification using TLC | | | | Verification using CADP | |
|---|---|---|---|---|---|---|
| | Concrete Spec. | | Abstract Spec. | | Quotient LTS | |
| | States | Time(s) | States | Time(s) | States | Time(s) |
| 3 | 395K | 157 | 14 | 15 | 14 | 12 |
| 4 | 27.285K | 1.080 | 32 | 23 | 20 | 20 |
| 5 | 113.285K | >36.000 | 82 | 64 | 47 | 42 |

abstraction reduces considerably the complexity of verification of C implementations.

When TLC reports that a transition violates the implementation formula $Spec \Rightarrow Abstract_instance!Spec$, there is an error either in the concrete specification, the abstract specification, or the refinement mapping function. The trace given by TLC can help to determine which one of those is the case. We use our tool to translate this trace in C and get the C execution sequence that leads to the error.

## V. Related work

Predicate abstraction [10] is a technique to abstract a program so that only the information about the given predicates are preserved. This technique is being used in SLAM [11], BLAST [12] and MAGIC [13]. Their approach has been shown to be very effective on specific application domains such as device drivers programming. SLAM uses symbolic algorithms, while BLAST is an on-the-fly reachability analysis tool. The Magic tool use LTS a specification formalism, and weak simulation as a notion of conformance of a system and its abstract specification.

These tools are applied to C programs and use automated theorem prover to construct the abstraction of the C program. The difficulty of these refinement-based approaches is that performing a refinement proofs between an abstract and a refined model require non trivial human effort and expertise in theorem proving to get the prover to discharge the proof obligations. SLAM cannot deal with concurrency, BLAST cannot handle recursion.

Besides predicate abstraction, several verification techniques for C programs have been proposed. CBMC [14] is a bounded model checker for ANSI C programs which translates a program into a propositional formula (in Static Single Assignment form), which is then fed to a SAT solver to check its satisfiability. CBMC explores program behavior exhaustively but only up to a given depth.

Compared to previous related works that use an over-approximation of the code implementation which is sound, our approach is based on constructing an executable abstract model, that can be expressed using TLA+ or by constructing the quotient LTS. Moreover, TLA+ is a logic that can express safety and liveness properties unlike SLAM, BLAST and CBMC which have limited support for concurrent properties as they only check safety properties.

## VI. Conclusion and Future Work

We have defined an operational semantics of a TLA+ specification in terms of a STSs. We redefined the semantics of refinement between a high-level (concrete) and a low-level (abstract) TLA+ specifications using STSs and we illustrated a way to automatically construct a quotient STS from the concrete specification by extending the TLC model checker. We applied all these notions for verifying C programs. Experimental results show that verifying properties on the abstract model reduces considerably the complexity of the verification process.

As future work, we plan to extend this work on several interesting directions. We would like to generate TLA+ and MCL properties from the ACSL [15] specification language used in Frama-C. We envisage 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. Finally, we aim to use the TLA+ proof system [16] to prove refinement between a concrete and abstract specifications.

## References


[1] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, Model checking. Cambridge, MA, USA: MIT Press, 1999.

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

[3] L. Leslie, "The Temporal Logic of Actions," ACM Trans. Program. Lang. Syst., vol. 16, no. 3, 1994, pp. 872–923.

[4] Z. Manna and A. Pnueli, The Temporal Logic of Reactive and Concurrent Systems. New York, NY, USA: Springer-Verlag New York, Inc., 1992.

[5] M. Abadi and L. Lamport, "The Existence of Refinement Mappings," Theor. Comput. Sci., vol. 82, no. 2, 1991, pp. 253–284.

[6] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2011: a toolbox for the construction and analysis of distributed processes," International Journal on Software Tools for Technology Transfer, vol. 15, no. 2, 2013, pp. 89–107.

[7] R. Mateescu and D. Thivolle, "A Model Checking Language for Concurrent Value-Passing Systems," in Proceedings of the 15th International Symposium on Formal Methods. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 148–164.

[8] A. Methni, M. Lemerre, B. Ben Hedia, S. Haddad, and K. Barkaoui, "Specifying and Verifying Concurrent C Programs with TLA+," in Formal Techniques for Safety-Critical Systems, C. Artho and P. C. lveczky, Eds. Springer, 2015, vol. 476, pp. 206–222.

[9] A. S. Tanenbaum, Modern Operating Systems, 3rd ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2007.

[10] S. Graf and H. Saïdi, "Construction of Abstract State Graphs with PVS," in Proceedings of the 9th International Conference on Computer Aided Verification. London, UK, UK: Springer-Verlag, 1997, pp. 72–83.

[11] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of c programs," in Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation, ser. PLDI '01. New York, USA: ACM, 2001, pp. 203–213. [Online]. Available: http://doi.acm.org/10.1145/378795.378846

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

[13] S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith, "Modular Verification of Software Components in C," IEEE Trans. Software Eng., vol. 30, no. 6, 2004, pp. 388–402.

[14] E. Clarke, D. Kroening, and F. Lerda, "A Tool for Checking ANSI-C Programs," in TACAS, K. Jensen and A. Podelski, Eds., vol. 2988. Springer, 2004, pp. 168–176.

[15] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto, ACSL: ANSI/ISO C Specification Language, version 1.4, 2009, [retrieved: October, 2015].

[16] D. Cousineau, D. Doligez, L. Lamport, S. Merz, D. Ricketts, and H. Vanzetto, "TLA+ Proofs," in 18th International Symposium on Formal Methods - FM 2012, D. Giannakopoulou and D. Méry, Eds., vol. 7436. Paris, France: Springer, 2012, pp. 147–154.


Copyright (c) IARIA, 2015. ISBN: 978-1-61208-438-1 