From Concurrent Programs to Simulating Sequential Programs: Correctness of a Transformation

Allan Blanchard
Univ. Orléans,
INSA Centre Val de Loire
LIFO EA 4022
45067 Orléans, France

Frédéric Loulergue
Northern Arizona University
School of Informatics
Computing and Cyber Systems
Flagstaff, USA

Nikolai Kosmatov
Software Reliability Laboratory
CEA, LIST, PC 174
91191 Gif-sur-Yvette, France

FRAMA-C is a software analysis framework that provides a common infrastructure and a common behavioral specification language to plugins that implement various static and dynamic analyses of C programs. Most plugins do not support concurrency. We have proposed CONC2SEQ, a FRAMA-C plugin based on program transformation, capable to leverage the existing huge code base of plugins and to handle concurrent C programs.

In this paper we formalize and sketch the proof of correctness of the program transformation principle behind CONC2SEQ, and present an effort towards the full mechanization of both the formalization and proofs with the proof assistant COQ.

1 Introduction

FRAMA-C [10, 9] is a framework for static and dynamic analysis of C programs. It offers a common infrastructure shared by various plugins that implement specific analyses, as well as a behavioral specification language named ACSL [2]. Developing such a platform is a difficult and time-consuming task. As most existing FRAMA-C plugins do not support *concurrent* C code, extending the current platform to handle it is an interesting and promising work direction.

Motivated by an earlier case study on deductive verification of an operating system component [3], we have proposed a new plugin, named CONC2SEQ [4], that allows FRAMA-C to deal with concurrent programs. In order to leverage the existing plugins, we designed CONC2SEQ as a code transformation tool. For *sequentially consistent* programs [12], a concurrent program can be simulated by a sequential program that produces all interleavings of its threads.

To ensure that the proofs and analyses conducted using CONC2SEQ are correct, we need to assure that the transformation preserves the semantics of programs. The contribution of this paper presents the proof of correctness of the code transformation principle used in CONC2SEQ.

The verification of the transformation is done for simplified languages that capture the interesting property with respect to validity, in particular memory accesses and basic data and control structures (both sequential and parallel). We formalize the source (parallel) language as well as the target (sequential) language and formally define the transformation on these languages.

In these languages, we do not consider all control structures of the C language but only simple conditionals and loops (`goto` and `switch` are not part of the considered languages). The C assignments are decomposed into three simpler constructs: local assignments that do not incur access to the global memory, reading of the global memory (one location at a time), and writing into the global memory (one location at a time). An expression can only be composed of constants, basic operations and local variables. Procedure calls are allowed but recursion is not. There is no dynamic memory allocation.

In the remaining of this report, we present first the considered source and target languages as well as their formal semantics (Section 2). Then we describe the transformation (Section 3). Section 4 is devoted to the equivalence relation between states of the source program and states of the transformed program, and its use for the proof of correctness of the proposed transformation. We discuss an ongoing effort to mechanize the formalization and proof with the interactive theorem prover COQ in Section 5. Finally, we position our contribution with respect to the literature in Section 6 and conclude in Section 7.

2 Considered Languages

2.1 Syntax and Program Definition

We consider an enumerable set of memory locations $\mathbb{L}$. We do not support dynamic memory allocation: the memory locations manipulated by a program are thus known before the beginning of the execution. A size is associated to each allocated location, i.e. the number of values that can be stored at this memory location. A location l can be seen as an array in C whose first element is $l[0]$ and whose address is l.

The set of values that can be assigned to variables is written $\mathscr{V}$ and is the union of memory locations ($\mathbb{L}$), integers ($\mathbb{Z}$) and booleans ($\mathbb{B}$). We assume that different values of the language take the same amount of memory.

We write $\mathscr{X}$ for the set of local variables. In the remaining of the paper, for a set A whose elements are written a, A^* is the set of finite sequences of elements of A and $\overline{a}$ will denote an element of A^*, i.e. a sequence of elements of A. Expressions are defined as follows:

$$\begin{array}{rcll} \mathscr{V} \ni v & ::= & n \mid l \mid b & n \in \mathbb{Z},\ l \in \mathbb{L},\ b \in \mathbb{B} \\ e & ::= & v \mid x \mid op(\overline{e}) & x \in \mathscr{X} \end{array}$$

We do not define the set of operators here: it is a usual set of arithmetic and boolean operations. It is however necessary to emphasize that these operators do not allow pointer arithmetic. The only provided operation on memory locations is comparison. Expressions cannot produce side-effects. In the remaining of the paper, expressions will be denoted by e and variants.

Sequential language. A sequential program is defined as a sequence of procedures, by convention the first one being the main procedure. A procedure is defined by its name, its parameters (that are a subset of local variables) and the sequence of instructions that form its body:

$$\begin{array}{lcll} proc & ::= & m(\overline{x})c & m \in Name \\ instr & ::= & x := e & \text{local assignment} \\ & \mid & x[y] := e & \text{writing to the heap} \\ & \mid & x := y[e] & \text{reading from the heap} \\ & \mid & \textbf{while } e \textbf{ do } c \quad \mid \quad \textbf{if } e \textbf{ then } c \textbf{ else } c & \\ & \mid & m(\overline{e}) & \text{procedure call} \\ \mathscr{C} \ni c & ::= & \{\} \mid instr;c & \end{array}$$

where $Name$ is the set of valid procedure names. **select**, **interleavings**, and names built from $\mathbb{Z}$ are all reserved names. $\mathscr{C}$ is the set of instruction lists, i.e. program code.

The language includes the usual primitives in a small imperative language: sequence of instructions (we will write $\{instr_1;instr_2\}$ instead of $instr_1;instr_2;\{\}$), conditionals, loops. Assignment is decomposed into three distinct cases: assignment of a local variable with the value of an expression, writing the value of an expression to the heap, and reading a value from the heap to a local variable. Expressions cannot contain reads from memory, nor procedure calls. A C assignment containing several accesses to the heap should therefore be decomposed into several reads into local variables and an assignment of an expression to a local variable, and finally, if necessary, a write to the heap from a local variable. Procedures can be called using the classical syntax $m(\overline{e})$ where $\overline{e}$ is the list of expressions passed in arguments. Arguments are passed by value.

A sequential program $prog_{seq}$ is fully defined by:

- the list of its procedures (the main one taking no parameter),
- a list of allocated memory locations with their associated sizes (positive numbers).

$$\begin{array}{rcl} memory & ::= & [(l_1, size_{l_1}); \ldots ; (l_m, size_{l_m})] \\ prog_{seq} & ::= & \overline{proc}\ memory \end{array}$$

Parallel language. A parallel program can be executed by any strictly positive number of threads. There is no dynamic creation of threads. During the execution of a parallel program the number of threads remains constant, given by a specific parameter of each execution. Let $\#tid$ denote this static number of threads.

$\mathbb{T}$ is the set of thread identifiers. We identify $\mathbb{T}$ with $\mathbb{N}$ seen as subset of $\mathbb{Z}$. An element of $\mathbb{T}$ is thus a *value* for both languages. A parallel program can use any of the sequential program constructs. In addition, it can contain the instruction **atomic**(c) that allows to run a sequence of instructions c atomically. In such a code section, no thread, other than the one that initiated the execution of the atomic block, can be executed.

A parallel program $prog_{par}$ is fully defined by:

- the list of its procedures,
- a list of allocated memory locations in the shared memory with their associated sizes,
- a mapping from thread identifiers to defined procedure names, defining the main procedure of each thread.

$$prog_{par} \ ::= \ \overline{proc}\ memory\ mains \qquad \text{where } mains : \mathbb{T} \to Name$$

For a program $prog$ (either sequential or parallel), mem_{prog} denotes the allocated memory of the program. This association list is also considered as a function, therefore $mem_{prog}(l)$ denotes the size allocated for memory location l, if defined. $procs_{prog}$ denotes the sequence of procedures of the program. For a parallel program $mains_{prog}$ is the mapping from $\mathbb{T}$ to *Name*, and for a sequential program $main_{prog}$ is the main procedure name. For a name m and a program $prog$, $body_{prog}(m)$ denotes the body of the procedure named m in the program $prog$. If it is clear from the context $prog$ may be omitted.

Comparison with the concurrent C of the FRAMA-C plugin. For sequential programs, the simplifications with respect to the subset of C handled by our CONC2SEQ plugin are essentially that we do not support pointer arithmetic, the expressions containing several memory reads or procedure calls should be decomposed, and we support only the "most structured" control structures. The typing is also very basic: variables and heap locations accept any type of values (integers, booleans, memory locations) and the type of expressions is checked dynamically by the semantic rules if necessary (for example the expression that is a condition of a loop or conditional should evaluate to a boolean value).

In C11, sequentially consistent concurrent atomic operations are often described by an equivalent sequential C program that is supposed to be atomically executed. In our FRAMA-C plugin, such operations are specified using ACSL and their calls placed into atomic sections. In the small imperative parallel language presented above, we could use the same technique: implement atomic operations as

```
atomic_transfer(l1, l2){
  atomic(v1 := l1[0];
         l2[0] := v1;)
}
```

```
sim_atomic_2(tid){
  tmp := &l1; l1  := tmp[tid];
  v1 := l1[0];
  tmp := &v1; tmp[tid] := v1;
  tmp := &l2; l2  := tmp[tid];
  tmp := &v1; v1  := tmp[tid];
  l2[0] := v1;
  tmp := pct; tmp[tid] := 4;
}
```

Figure 1: Atomic transfer of the value of a global variable to another, and the resulting simulating method for the corresponding atomic instruction.

their sequential counterparts and put their calls into atomic blocks. For example, we illustrate the atomic transfer of the value of an global variable to another one in Figure 1. It is composed of two instructions that are executed in a single atomic step. The resulting simulating code will be commented later.

In our case studies, the concurrent C programs do not need to know the number of threads, and actually do not depend on the number of threads except for one specific feature: global variables that are *thread local*. This kind of variables are in shared memory, but each thread has its own independent copy. This is particularly useful to have thread dedicated copies of global variables such as `errno`. In this case, in our memory model it would mean that the number of memory locations called `errno` would be dependent on the number of threads. The set of allocated memory locations does not depend on the number of threads.

If we want to model a procedure `f` that uses a thread local variable `tlv` we can define in our parallel language a procedure f that takes an additional argument *tlv* and use, for each thread, a different main procedure calling f with a specific allocated memory location passed to argument *tlv*.

However the set of allocated memory locations (as well as the number of different main procedures) is not dependent on the number of running threads. We can then imagine to have a kind of extended parallel language which could contain symbolic names for thread local variables and a pre-processor that, for a specific value of #*tid*, would generate programs of the proposed parallel language (generating as many memory locations and main procedures as necessary). As the transformation presented in Section 3 from the proposed parallel language to the proposed sequential language also depends on #*tid*, we do not consider this aspect to be a limitation of our modelling approach. These modelling choices allow to keep both languages simple and representative.

2.2 Semantics

For a sequential program, or a thread, the local environment ρ is a partial function from local variables to values: $\rho : \mathscr{X} \rightharpoonup \mathscr{V}$. The set of local environments is written $\mathscr{E}$. $\varnothing$ denotes the empty environment, i.e. the function undefined everywhere.

For both the sequential and the parallel languages, a heap $\eta : \mathbb{L} \rightharpoonup \mathbb{N} \rightharpoonup \mathscr{V}$ is a partial function from memory locations that returns a partial function from indices to values, thus essentially defining an array indexed from 0. $\mathscr{H}$ is the set of heaps. For a defined memory location, the associated partial function is defined continuously for indices from 0 to a fixed size.

A *local execution context* is composed of the name of the procedure being executed, a local environment and the code that remains to execute. The set of local execution contexts is $\mathscr{L} = Name \times \mathscr{E} \times \mathscr{C}$. A call stack is defined as a sequence (stack) of local execution contexts: $s \in \mathscr{S} = \mathscr{L}^*$.

The states of sequential and parallel programs are respectively:

$$\Sigma_{seq} \;=\; \mathscr{S} \times \mathscr{H} \qquad\qquad \Sigma_{par} \;=\; (\mathbb{T} \rightharpoonup \mathscr{S}) \times \mathscr{H}$$

For a parallel state $\sigma_{par} \in \Sigma_{par}$, we denote by $stacks_{\sigma_{par}}$ the first component of the state, i.e. the mapping from thread identifiers to stacks of local execution contexts. We omit the index σ_{par} when it is clear from the context.

Initial contexts and states. The initial execution stack is $[(main,\ \varnothing,\ body(main))]$ for a sequential program. For a parallel program, the initial context of a thread $t \in \mathbb{T}$ is $[(mains(t),\ \varnothing,\ body(mains(t)))]$. For a sequential program, an initial state is thus: $([(main,\ \varnothing,\ body(main))],\ \eta^{init}_{seq})$. For a parallel program, an initial state is $(stacks_{init},\ \eta^{init}_{par})$ where $\forall t \in \mathbb{T}.\ stacks_{init}(t) = [(mains(t),\ \varnothing,\ body(mains(t)))]$.

An initial heap η^{init}_{seq} should satisfy the memory allocation defined by a sequential program, i.e. if $(l, size) \in mem$ then $\eta^{init}_{seq}(l)(i)$ is defined for all $0 \le i < size$. In addition, the values contained in such a memory location cannot be themselves memory locations (but they can be any other values). The same constraints hold for an initial heap of a parallel program.

Final states and safe execution The final state of a sequential program is such that $\exists \eta.\ \sigma^{final}_{seq} = ([],\ \eta)$ and the final state of a parallel program is such that $\exists \eta.\ \sigma^{final}_{par} = (stacks,\ \eta)$ with $\forall t \in \mathbb{T}.\ stacks(t) = []$.

We define a blocking state as a non final state reached from an initial state such that no semantic rule can make the execution progress. A *safe program* is a program that does not reach a blocking state from any initial state. In particular, a safe program can have non-terminating executions.

Actions The sequential programs produce 5 basic actions: silent action, procedure call, procedure return, memory reading, memory writing. For parallel programs, the atomic block structure requires to have an action list as a possible action:

$$a_{seq} \;::=\; \tau \mid \textbf{call}\ m\ \overline{v} \mid \textbf{return}\ m \mid \textbf{read}\ l\ n\ v \mid \textbf{write}\ l\ n\ v \qquad\qquad a_{par} \;::=\; a_{seq} \mid \textbf{atomic}\ \overline{a_{seq}}$$

Execution traces are action lists for sequential programs and lists of events, i.e. pairs of thread identifier and action, for parallel programs.

Operational semantics The operational semantics of sequential programs is defined in Figure 2 (rules for loops and conditionals are omitted, see [5]). A judgement of the sequential semantics has the following form: $\mathscr{P} \vdash s,\ \eta \xrightarrow{a_{seq}} s',\ \eta'$, meaning that a new state $(s',\ \eta')$ is reached from the state $(s,\ \eta)$ and this execution step produces an action a_{seq}. $\mathscr{P}$ is a program definition. We write $\mathscr{P} \vdash s,\ \eta \xrightarrow{\overline{a_{seq}}}{}^{*}\ s',\ \eta'$ for the reflexive and transitive closure of the relation defined by the inference system of Figure 2.

We use the following notations: $l_1 +\!\!+\, l_2$ is the concatenation of two sequences/lists. To add an element on top (i.e. on the left) of a sequence, we use the separator ";" for sequences of instructions, and the separator "$\cdot$" for sequences of local contexts (stacks). $|l|$ is the length of the sequence l. We write $x \in l$ to denote that x is an element of the sequence l, and by abuse of notation, that x is a component of a tuple in the list of tuples l. $f[a \mapsto b]$ is the function f' such that $f'(a) = b$ and for all elements a' different from a, we have $f'(a') = f(a')$. For two sequences $\overline{a}$ and $\overline{b}$ of equal length, we write $f[\overline{a} \mapsto \overline{b}]$ instead of $f[a_1 \mapsto b_1] \ldots [a_n \mapsto n_n]$. Thus $\rho[x \mapsto v]$ denotes an update of variable x with value v in environment ρ while $\eta[(l,o) \mapsto v]$ denotes an update at offset o of memory location l with value v in heap η. When it is the empty environment that is updated, we omit it.

$$\begin{array}{lll}
\mathscr{P} \vdash (m,\ \rho,\ (x := e;\ c)) \cdot s,\ \eta & \xrightarrow{\tau} & (m,\ \rho[x \mapsto v],\ c) \cdot s,\ \eta \\
[\textbf{assign}] & \text{if } [\![e]\!]_\rho = v & \\[2ex]
\mathscr{P} \vdash (m,\ \rho,\ (x[e_o] := e_v;\ c)) \cdot s,\ \eta & \xrightarrow{\textbf{write}\ l\ o\ v} & (m,\ \rho,\ c) \cdot s,\ \eta[(l,o) \mapsto v] \\
[\textbf{write}] & \text{if } [\![e_v]\!]_\rho = v,\ [\![e_o]\!]_\rho = o,\ \rho(x) = l,\ o < mem(l) & \\[2ex]
\mathscr{P} \vdash (m,\ \rho,\ (x := y[e_o];\ c)) \cdot s,\ \eta & \xrightarrow{\textbf{read}\ l\ o\ v} & (m,\ \rho[x \mapsto v],\ c) \cdot s,\ \eta \\
[\textbf{read}] & \text{if } [\![e_o]\!]_\rho = o,\ \rho(y) = l,\ o < mem(l),\ \eta(l)(o) = v & \\[2ex]
\mathscr{P} \vdash (m,\ \rho,\ (m'(\overline{e});\ c)) \cdot s,\ \eta & \xrightarrow{\textbf{call}\ m'\ \overline{v}} & (m',\ [\overline{x} \mapsto \overline{v}],\ c_{m'}) \cdot (m,\ \rho,\ c) \cdot s,\ \eta \\
[\textbf{call}] & \text{if } m'(\overline{x})c_{m'} \in \mathscr{P},\ |\overline{x}| = |\overline{e}|,\ \overline{[\![e]\!]_\rho} = \overline{v},\ m' \notin s & \\[2ex]
\mathscr{P} \vdash (m,\ \rho,\ []) \cdot s,\ \eta & \xrightarrow{\textbf{return}\ m} & s,\ \eta \\
[\textbf{return}] & & \\[2ex]
\mathscr{P} \vdash (m,\ \rho,\ (\textbf{select}_{\#tid}(tid,\ pc);\ c)) \cdot s,\ \eta & \xrightarrow{\textbf{call select}\ [l_{tid},\ l_{pc}]} & (m,\ \rho,\ c) \cdot s,\ \eta[(l_{tid},0) \mapsto t] \\
[\textbf{select}] & \text{if } [\![tid]\!]_\rho = l_{tid},\ [\![pc]\!]_\rho = l_{pc},\ 0 \le t < \#tid,\ \eta(l_{pc})(t) \neq 0 &
\end{array}$$

Figure 2: Operational semantics of sequential programs

$[\![e]\!]_\rho$ corresponds to the evaluation of expression e in local environment ρ. We omit the definition of this evaluation that is classic. For example for a variable x, $[\![x]\!]_\rho = \rho(x)$.

This semantics is rather usual, but condition $m' \notin s$ in rule [**call**] forbids recursive procedure calls. Moreover there is a special procedure call: $\textbf{select}_{\#tid}(tid,\ pc)$. This is the only non-deterministic rule of the sequential language. It selects randomly a value t between 0 and $\#tid$ (excluded), such that pc is a memory location which is defined at index t and contains a value different from 0 (reserved for terminated threads). The memory location tid is updated with this value t. This procedure call will be used in the simulation to model the change of current thread. Note that this procedure is not supposed to be called in parallel programs.

$$\begin{array}{lll}
\mathscr{P},\ \#tid \vdash stacks,\ \eta & \xrightarrow{(t,a_{seq})} & stacks[t \mapsto s'],\ \eta' \\
[\textbf{seq}] & \text{if } \mathscr{P} \vdash stacks(t),\ \eta \xrightarrow{a_{seq}} s',\ \eta' \text{ and } 0 \le t < \#tid & \\[2ex]
\mathscr{P},\ \#tid \vdash stacks,\ \eta & \xrightarrow{(t,\textbf{atomic}\ \overline{a_{seq}})} & stacks[t \mapsto (m,\ \rho',\ c) \cdot s],\ \eta' \\
[\textbf{atomic}] & \text{if } \mathscr{P} \vdash [(m,\ \rho,\ c_{atomic})],\ \eta \xrightarrow{\overline{a_{seq}}}{}^{*} [(m,\ \rho',\ [])],\ \eta' & \\
 & \text{where } stacks(t) = (m,\ \rho,\ (\textbf{atomic}(c_{atomic});\ c)) \cdot s \text{ and } 0 \le t < \#tid &
\end{array}$$

Figure 3: Operational semantics of parallel programs

Figure 3 presents the semantics of parallel programs. A judgement of this semantics have the following form: $\mathscr{P},\ \#tid \vdash stacks,\ \eta \xrightarrow{(t,a_{par})} stacks',\ \eta'$, where we recall that $\#tid$ is a strictly positive number of threads.

A thread t is selected such that $0 \leq t < \#tid$ and t has code to execute. If the first instruction of t is not an atomic block, then the state is reduced using the semantics of the sequential language. In this case the whole shared heap is given as the heap of the sequential reduction. The action of the sequential reduction is combined to the thread identifier t to form the event of the parallel reduction.

If the first instruction of t is an atomic block, then we use the sequential semantics to reduce the whole block. As we reduce the whole instruction sequence without allowing for a change of thread, the execution of this sequence is indeed atomic. The nesting of atomic blocks is not allowed: our semantics would be stuck in this case.

3 Program Transformation

Let us consider a parallel program $\overline{procs}$ *memory mains*. The memory of the simulating sequential program contains: *memory*, a fresh memory location `pct` of size $\#tid$, a fresh memory location `ptid` of size 1, for each procedure m a fresh memory location $\texttt{from}(m)$ of size $\#tid$ (with $\texttt{from} : Name \to \mathbb{L}$). *memory* will be shared by the threads. The array `pct` contains for each thread identifier t (therefore at index t) the simulation of the program counter of the thread identified by t, while `ptid` contains the identifier of the current running thread. $\texttt{from}(m)$ is used to manage the return of calls to m in the simulating code.

The three assignment instructions are supposed to be atomic. For loops and conditionals, the evaluation of the condition is supposed to be atomic. The transformation essentially translates each *atomic instruction* of each procedure of the parallel program into one *procedure* of the simulating sequential program. This procedure has a parameter tid that is supposed to be the identifier of the active thread running the instruction. In the remaining of the paper, variables written is $\mathsf{sans\text{-}serif\ font}$ are fresh variables not used in the input parallel program, but that we need to implement in the simulating sequential program, such as tid.

We assume that the input parallel program is labeled: each instruction *instr* is labeled by two values of $\mathbb{Z} \setminus \{0\}$ (0 is a label that indicated termination), such that the first one, denoted ℓ, is a unique label in the program definition, and the second one, denoted ℓ_{next}, is the label of the instruction that follows the current instruction in the program text (for example the label of the next instruction of a conditional is the instruction that follows the conditional, not the label of one of the branches). We write $instr^{\ell}_{\ell_{next}}$ for such a labeled instruction. One important point is that the label ℓ_{next} of the last instruction of *each* procedure is a label distinct from all the labels in the program. $begin(m)$ is a function that returns the label of the first instruction of the body of procedure m. $end(m)$ returns the label ℓ_{next} of the last instruction of the procedure body. If the body is empty, both functions returns a label distinct from all other labels in the program.

For each local variable x of the program (uniquely identified by the name m of the procedure in which it appears and its name x), including procedure formal parameters, we need a fresh memory location $\&_m x$ of allocated size $\#tid$ (we omit m in the remaining of the paper), so that each simulated thread has a copy of what was a local variable in the parallel program.

We detail how the transformation proceeds on an example instruction: $(x := y+1)^{\ell}_{\ell_{next}}$. This instruction will be transformed into a procedure named ℓ with parameter tid (we assume a coercion *toName* from $\mathbb{Z}$ to *Name*, and we omit it most of the time). y is simulated by the array $\&y$. As reads from the heap are not allowed in expressions, in the simulated code we first need to read the value from $\&y$. We write this sequence of instructions $load(y)$ defined as $\mathsf{tmp} := \&y;\ y := \mathsf{tmp}[\mathsf{tid}]$. Note that after this sequence of instructions, variable y is defined, therefore the original expression can be used as is. The original assignment however should be translated too as x is simulated by an array $\&x$. We translate it to: $\mathsf{tmp} := \&x$; $\mathsf{tmp}[\mathsf{tid}] := y+1$. Finally we update the program counter of the running thread, so the full translation of the instruction is:

$$\ell(\mathsf{tid})\{\ \mathsf{tmp} := \&y;\ y := \mathsf{tmp}[\mathsf{tid}];\ \mathsf{tmp} := \&x;\ \mathsf{tmp}[\mathsf{tid}] := y+1;\ \mathsf{tmp} := \mathtt{pct};\ \mathsf{tmp}[\mathsf{tid}] := \ell_{next}\ \}$$

The generalization to an arbitrary $x := e$ is just that we "load" all the variables of e before using e. Reading from the heap and writing to the heap are translated in a very similar way. Figure 1 provides a more complex example with the simulating code of the atomic memory transfer.

Both conditional and loops are translated into a procedure that evaluates the condition and then updates the program counter to the appropriate label. For example, if the condition of a conditional is true then the program counter is updated to the label of the first instruction of the "then" branch of the original conditional, if this branch is non-empty, otherwise the label used is the label of the instruction that follows the original conditional.

Each procedure call is translated into one procedure that passes the values to parameters and updates the program counter to the first instruction of the body original procedure (label $begin(m)$ for a call to m). Also for each procedure m we generate an additional procedure, named $end(m)$, that manages the return of calls to m. This procedure should be able to update the program counter to the instruction that follows the call. To be able to do so for any call, this return procedure should use a label previously stored at memory location $\mathtt{from}(m)$ by the generated procedure that prepares the call:

$$end(m)(\mathsf{tid})\{\ \mathsf{tmp} := \mathtt{from}(m);\ \mathsf{aux} := \mathsf{tmp}[\mathsf{tid}];\ \mathsf{tmp} := \mathtt{pct};\ \mathsf{tmp}[\mathsf{tid}] := \mathsf{aux}\ \}$$

One procedure is generated for each atomic block. Each instruction in the block is generated in a similar way as previously described but no update to the program counter is done, conditionals and loops keep their structure and their blocks are recursively translated in the atomic fashion. Procedure calls are *inlined* and the body of the called procedure is translated in the atomic fashion. It is necessary that procedures are not recursive for this inlining transformation to terminate.

Finally the main procedure of the simulating sequential program, named **interleavings**, is generated (Figure 4). It has basically two parts: in the first part (denoted by c_{init}) each program counter is updated to the identifier of the first instruction of the main procedure of the considered thread. c_{init} places the value at location $\mathtt{from}(mains(t))$ to 0 to stop the execution when the main procedure ends. c_{init} also initializes the local variable $\mathsf{terminated}$, that indicates if all threads are terminated, to **false**. We suppose that there is at least one thread with a main procedure to execute. If it were not the case, we would initialize it to **true**. The second part is the main simulating loop: if there are still threads to run, a thread identifier of an active thread is chosen (call to **select**, instruction named c_{select}), then the value ℓ of the program counter for this thread is read and a switch (it is implemented as nested conditionals, we use it here for the ease of presentation) calls the appropriate procedure named ℓ (sequence of instructions named c_{sim}). The body of this loop ends by updating the flag that indicates if there are still running threads (sequence of instructions named $c_{termination}$).

4 Correctness

To state the correctness theorem, we need two notions of equivalence: state equivalence, relating states of the input parallel program and states of the simulating sequential program, and trace equivalence that relates traces generated by the input parallel program to traces generated by the simulating sequential program.

```
interleavings(){
  // $c_{init}$
  tmp:=pct;
  $\forall t \in [0,\#tid[$, tmp[$t$]:=$begin(mains(t))$;
  $\forall t \in [0,\#tid[$, tmp:=from($mains(t)$); tmp[$t$] := 0;
  terminated:=false;
  while ¬terminated do {
    // $c_{select}$
    select$_{\#tid}$(ptid, pct);
    // $c_{sim}$
    tmp:=ptid; tid:=tmp[0]; tmp:=pct; aux:=tmp[tid];
    switch aux is { $\ell : toName(\ell)$(tid) }
    // $c_{termination}$
    terminated:=true; tmp:=0;
    while tmp < $\#tid$ do {
      if pct[tmp] ≠0 then { terminated:=false } else { };
      tmp:=tmp+1;
    }
  }
}
```

Figure 4: Main procedure of the simulating sequential program

4.1 Equivalence of states and traces

We note σ_{sim} the sequential program state $(s_{sim},\ \eta_{sim})$ of the simulation of a safe parallel program in a state $\sigma_{par} = (s_{par},\ \eta_{par})$. In η_{sim}, we distinguish two disjoint parts η_{sim}^{par} that replicates η_{par} and η_{sim}^{sim} the addresses that simulate the local variables of s_{par}. This second part also includes `pct`, `ptid`, and the addresses `from`(m). The syntax $\eta_{sim}^{sim}[t]$ allows to partially apply η_{sim}^{sim} for the thread t to select the part that simulates t. So the function $\eta_{sim}^{sim}[t](l)$ is $\eta_{sim}(l,t)$. We define state equivalence as follows:

$$\eta_{par} = \eta_{sim}^{par} \tag{1}$$

$$\forall t \in \mathbb{T},\ \rho \in stacks(t),\ x \in \mathscr{X}\ .\ \rho(x) = v \Longrightarrow \eta_{sim}^{sim}[t](\&x) = v \tag{2}$$

$$\forall t \in \mathbb{T},\ ctx \in \mathscr{L},\ s \in \mathscr{S}.\ stacks(t) = ctx \cdot s \Longleftrightarrow \eta_{sim}^{sim}[t](\texttt{pct}) = \text{NEXT}(ctx) \tag{3a}$$

$$\forall t \in \mathbb{T}.\ stacks(t) = [] \Longleftrightarrow \eta_{sim}^{sim}[t](\texttt{pct}) = 0 \tag{3b}$$

$$\forall t \in \mathbb{T}.\ \text{WF\_STACK}(stacks(t),\ \eta_{sim}^{sim}[t]) \tag{4}$$

$$\frac{\begin{array}{l} s_{sim} = (\textbf{interleavings},\ \rho_{sim},\ \{\textbf{while}\ \neg\text{terminated}\ \textbf{do}\ (c_{select} +\!\!+\, c_{sim} +\!\!+\, c_{termination})\}) \wedge \qquad (5) \\ \qquad (\rho_{sim}(\text{terminated}) = \textbf{true} \Longleftrightarrow \forall t \in \mathbb{T}.\ \eta_{sim}^{sim}[t](\texttt{pct}) = 0) \end{array}}{\sigma_{par} \sim \sigma_{sim}}$$

(1) expresses the fact that the original heap should be a sub-part of the simulating heap. For each thread t, (2) relates the content of every local variable x of t by the content of the global array $\&x$ in $\eta_{sim}^{sim}[t]$ that

simulates it.

Program counters must be correctly modeled, (3a) and (3b) express that each program counter must point on the NEXT instruction to execute by thread t if any (3a), 0 if not (3b). Call stacks must be correctly modeled by $\texttt{from}(m)$ (4). We refer to [5] for the formal definitions of NEXT, that returns the label of the next instruction to execute in a non-empty local execution context, and WF_STACK, that relates the call stacks of the parallel state with the labels at memory locations $\texttt{from}(m)$. Finally in condition (5), the equivalence is defined for simulating program states such that the next step to perform is the evaluation of the condition of the loop since the simulation of an instruction is the execution of this evaluation followed by the body of the loop.

The equivalence of traces is defined on filtered lists of actions generated by the semantics. In the simulating program executions, we ignore τ-actions and memory operations in η^{sim}_{sim}. We ignore all call to and return from simulating procedures except for calls to **select**, and procedures that simulate the start of a call and the return of a call.

4.2 Correctness of the simulation

Theorem 1 (Correct simulation)**.** *Let $prog_{par}$ be a safe parallel program, $prog_{sim}$ its simulating program, σ^{init}_{par} (resp. σ^{init}_{sim}) an initial state of $prog_{par}$ (resp. $prog_{sim}$).*

i. From σ^{init}_{sim}, we can reach, by the initialization sequence c_{init}, σ^{0}_{sim} equivalent to σ^{init}_{par}.

ii. For all σ_{par} reachable from σ^{init}_{par}, there exists an equivalent σ_{sim} reachable from σ^{0}_{sim} with an equivalent trace (Forward simulation).

iii. For all σ_{sim} reachable from σ^{0}_{sim}, there exists an equivalent σ_{par} reachable from σ^{init}_{par} with an equivalent trace (Backward simulation).

The proof of this theorem rely on two main observations. First, the parallel semantics is deterministic except for the choice of the thread, which is not an operation of the program. Equivalently, the only non-deterministic operation of the simulation is the call to **select**, that models the non-deterministic behavior of the parallel semantics. Second, once the parallel semantics has selected a thread, the reduction is delegated to the sequential semantics that is deterministic. The corresponding simulating code, that solves the program counter and execute the simulating procedure, is also deterministic. Now, if we prove the forward simulation for a transformation and the resulting code is deterministic, then we also prove the backward simulation, as pointed by [13, Def 5.]. More detailed proofs of theorem 1 can be found in the report [5].

We show that the initialization establish the equivalence (i.) by induction on traces. For the forward simulation (ii.), the induction is on the instructions, for the backward simulation (iii.), on the number of iterations of the interleaving loop.

Initialization An initial state of the simulation is:

$$((\textbf{interleavings},\ \rho_{sim},\ c_{init} +\!\!+ \{\textbf{while}\ \neg\mathsf{terminated}\ \textbf{do}\ (c_{select} +\!\!+ c_{sim} +\!\!+ c_{termination})\}),\ \eta_{sim})$$

As we suppose (by construction) that initially, $\eta^{par}_{sim} = \eta_{par}$ and that η^{sim}_{sim} contains correctly allocated simulation blocks for local variables and by the definition of a parallel initial state: $(stacks_{init},\ \eta^{init}_{par})$ such that $\forall t \in \mathbb{T}.\ stacks_{init}(t) = [(mains(t),\ \varnothing,\ body(mains(t)))]$ the parts (1) and (2) of the equivalence are verified. The idea is then to show that the execution of c_{init} correctly establish (3a), (3b), (4) and (5). In c_{init}, we first move each program counter to the first instruction of each main procedure, ensuring (3a) and (3b) , and then initialize the `from` address of each of these main procedures to ensure (4) (the base of the stack is correctly modeled). Finally, we initialize `terminated` to false, since each thread must, at least, return from its main, ensuring the (5). We have reached a state σ^0_{sim} equivalent to $stacks_{init}$, η^{init}_{par}.

Lemma 2 (Forward simulation on a single step)**.** *Let $prog_{par}$ be a safe parallel program and $prog_{sim}$ its simulating program, σ_{par} a parallel state that reaches σ'_{par} with an event (t, a_{par}), σ_{sim} equivalent to σ_{par}, there exists a trace tr equivalent to $[(t, a_{par})]$ that allows to reach σ'_{sim} equivalent to σ'_{par}.*

Sketch of proof. By the equivalence relation, we know that σ_{sim} is of the form:

$$((\ldots,\ \rho_{sim},\ \textbf{while}\ \neg\mathsf{terminated}\ \textbf{do}\ (c_{select} +\!\!+\, c_{sim} +\!\!+\, c_{termination})),\ \eta_{sim})$$

In the parallel semantics, we perform a step of reduction for the thread t, so its stack is not empty, and by (3a), we know that $\eta^{sim}_{sim}[t](\mathtt{pct}) \neq 0$, and consequently, by (5), $\rho_{sim}(\mathsf{terminated}) = \textbf{false}$. We get the simulating program state:

$$((\ldots,\ \rho_{sim},\ c_{select} +\!\!+\, c_{sim} +\!\!+\, c_{termination} +\!\!+\, \{\textbf{while}\ \neg\mathsf{terminated}\ \textbf{do}\ (c_{select} +\!\!+\, c_{sim} +\!\!+\, c_{termination})\}),\ \eta_{sim})$$

We then perform the reduction [**select**]. It generates an action **call select** $[l_{\mathtt{ptid}};\ l_{\mathtt{pct}}]$ (the first action of tr) that places t at memory location `ptid`, t being an allowed choice for **select** since $\eta^{sim}_{sim}[t](\mathtt{pct}) \neq 0$.

At this step, we perform a case analysis depending on the executed instruction and prove that the execution reach a state where the parts (1) to (4) of the equivalence are verified. Then, the execution of $c_{termination}$ updates the variable terminated by successively comparing the program counters to 0. As we maintained (3a) and (3b), we reach a state such that (5) is verified:

$$((\ldots,\ \rho_{sim},\ \{\textbf{while}\ \neg\mathsf{terminated}\ \textbf{do}\ (c_{select} +\!\!+\, c_{sim} +\!\!+\, c_{termination})\}),\ \eta''_{sim})$$

Moreover, actions generated during this loop are reads in η^{sim}_{sim} and τ-actions (that are filtered). We reach, from σ_{sim} equivalent to σ_{par}, a state σ'_{sim} equivalent to σ'_{par}, with a trace tr equivalent to $[(t, a_{par})]$. □

Lemma 3 (Backward simulation on a single step)**.** *Let $prog_{sim}$ be the simulating program of a safe parallel program $prog_{par}$, σ_{sim} a sequential state that reach σ'_{sim} with a trace $tr = (\textbf{call}\ \textbf{\textit{select}}\ [l_{\mathtt{ptid}};\ l_{\mathtt{pct}}]);tr'$, such that tr' does not contain call action to select, σ_{par} equivalent to σ_{sim}, there exists an action (t, a_{par}) such that $[(t, a_{par})]$ is equivalent to tr that allows to reach σ'_{par} equivalent to σ'_{sim}.*

Sketch of proof. Starting from $((\ldots,\ \rho_{sim},\ \{\textbf{while}\ \neg\mathsf{terminated}\ \textbf{do}\ (c_{select} +\!\!+\, c_{sim} +\!\!+\, c_{termination})\}),\ \eta_{sim})$, the simulation builds a trace $tr = (\textbf{call select}\ [l_{\mathtt{ptid}},\ l_{\mathtt{pct}}]);tr'$ so the condition is evaluated to **true** (else we would not execute the loop, and the first action of the trace would not be realized). We also know that there exists t such that $\eta^{sim}_{sim}(pct) \neq 0$. We know that in the original program, there exists t such that $stacks(t)$ is not empty, and we know the instruction *instr* that has to be performed by t.

As $prog_{par}$ is safe, it does not block, the instruction *instr* of t can be executed, so there exists a new parallel state σ'_{par}, reached with an action (t, a_{par}). By lemma 2, we know that there exists a simulated state $\sigma'_{sim?}$ equivalent to σ'_{par} reached from σ_{sim} with a trace tr_f equivalent to $[(t, a_{par})]$. This trace tr_f starts with an action **call select** $[l_{ptid}]$ equivalent to the one produced for tr and represents the execution of *instr* by t, that is also simulated by our program $prog_{sim}$. We can deduce that $\sigma'_{sim?} = \sigma'_{sim}$. As σ'_{par} is equivalent to $\sigma'_{sim?}$, it is also equivalent to σ'_{sim}. Moreover $tr = tr_f$, tr_f is equivalent to $[(t, a_{par})]$, so tr is equivalent to $[(t, a_{par})]$. □

The case analysis of each type of instruction is not presented since it is a lot a details (that can be found in [5]). The ideas are quite the same from a proof to another: we show that (2) ensures we correctly replicate the local variables in the simulating procedure, we deduce that the "actual" simulation instruction performs exactly the action that the original one (which maintains (1) and (2)), and finally we show that the program counter is correctly updated according to the next instruction ((3a) and (3b)). The only tricky part is in function call and returns where we have to ensure that `from` is updated with respect to the original stack update and maintains (4), for other instructions this part is trivial since the stack does not change and `from` is not updated.

5 Towards a Mechanized Proof of Correctness

We aim at mechanizing the proof of correctness using the proof assistant Coq. A first step to do so is to formalize both languages and their semantics, as well as the transformation. The current state of the development[1] includes this first step (about 3,000 lines of Coq, one third being proofs).

We have roughly 20% devoted to supporting definitions and results (about quite general data types and data structures used in the rest of the formalization), 50% to the syntax and semantics of the two programming languages (about half of it comes from another project with only slight modifications), the remaining 30% focusing on the formalization of the transformation and the statement of the correctness theorem.

The syntax and semantics of the languages have a rather usual formalization. As we seek reuse, we modeled the sequential semantics so that it is parametrized by a set of "external procedure" definitions as it is found in some programming languages where the signatures of some procedures are given but their implementation is done in a foreign language. Here some procedures are not defined in the programming language but are axiomatized by additional semantic rules. `select` is defined by such an external procedure definition.

One important difference between the program definitions on paper and in Coq, is that in the mechanized version, all should be explicit. For example, in Section 2, we leave implicit that procedure names should not be duplicated in the list of procedure definitions.

The validity of procedure calls is used to define a relation on procedures. For two procedures p_1 and p_2 of a program $\mathscr{P}$, we have $p_1 \prec p_2$ if the body of p_2 contains a call to p_1. To ensure that all procedures of a program are non-recursive, if is sufficient to require that $\prec$ is well-founded. This property is necessary for two reasons. First, our simplified way to simulate the call stack in the transformed code requires it. Second, Coq requires that all functions are *terminating*. It automatically checks the termination of recursive functions when the recursive calls are done on syntactical sub-terms of one of the arguments of the function. In other cases, a proof of termination should be given.

The next step of the mechanization is to define the equivalence between states: the properties about the uniqueness of procedure names and correct labelling are very important in this regard. The final step will be to prove the correctness.

6 Related Work

Many model checking tools for concurrent programs are based on code sequentialization. In [17], Qadeer and Wu present, for the C language, a transformation from parallel to sequential code that allows the use

[1] Available at `http://frederic.loulergue.eu/ftp/cconc2seq-0.1alpha.tar.gz`

of existing model checkers for sequential systems. This bounded model checking has been generalized to any context bounds with CSeq [11] and dynamic thread creation [6]. While bounded, such an approach is still efficient to find bugs in concurrent programs. Regarding code transformation, these approaches differ from ours since in each thread, functions are inlined in the main function, loops are unrolled and k copies of the global memory are kept for a bound of K thread context switching. To avoid creating these copies, allow dynamic memory allocation and improve performances, Fisher et al. [8] propose a lazy version of these tools called LazySeq that shows high performances on known benchmarks. Other authors choose to bound memory accesses instead of context switching [18].

While efficient to find bugs, these approaches are not suited to prove safety, which is the main reason we aimed at supporting the WP plugin of FRAMA-C. In [14], Nguyen et al. further generalize LazySeq to unbounded concurrent programs allowing safety checking. The approach for code generation is somehow dual to ours: instead of splitting original functions into smaller functions for each statement and adding the context switching management in an interleaving loop, context switching is modeled inside each function to obtain a behavior where each call to the function will execute a step of execution and then return (and where local variables become static). All these approaches consider a sequentially consistent memory model, as we do, while other work is aimed at supporting weaker behaviors as well, e.g. [19].

Why3 is a deductive verification tool that proposes Why-ML, a language for writing programs and assertions, and a verification condition generator as well as translations of these conditions as input to a wide variety of automated provers. Fortin and Gava [7] used program transformation to perform deductive proof of bulk synchronous parallel (BSP) programs. In this work, the original annotated parallel program is compiled into an equivalent sequential Why-ML program. The deductive proof is then performed using the original Why-ML VCGen, designed for sequential programs. The transformation is written and proved using the COQ proof assistant. If the software context is very close to our proposal, the parallelism models are very different and thus so are the transformations: A BSP program is a sequence of super-steps, and the parallelism occurs inside each super-step. Inside a super-step each thread computes using only the data it holds in memory, then communicates with other threads, but the results of these communications (by message passing) are not effective before the end of the super-step (that contains a synchronization barrier).

The way we transform code and specification makes the use of WP after the transformation closely related to Owicki-Gries method [16]. Indeed, for each instruction, we have to ensure that it is compatible with any state of the global system that can be reached at some program point. This property is modeled by a global invariant. Unlike [16], this compatibility is not verified by visiting the proof tree. Owicki-Gries method has been formalized in Isabelle/HOL [15] and one of its variants has been used for verification of operating systems [1]. So, even if it can generate a lot of verification conditions, it is still usable in practice for real-life code.

7 Conclusion

The contribution of this paper is the correctness proof of the principle of a code transformation used to verify concurrent C code through a sequential C program that simulates it, in the context of a sequentially consistent memory model. This proof is done under the assumption that the source program does not allocate memory and does not contain any recursive call. This proof has three main concerns: the heap of the source program should be correctly replicated in the transformed program; the local environments of the source program should be correctly simulated by the *global heap* of the transformed program; the execution context of the source program should be correctly modelled by the memory location that stores a kind of program counter and the memory locations that model a simplified call stack.

The proof relies on the fact that in a way the simulating code mimics the operational semantics of the concurrent program with its own sequential instructions, but in a simplified version (in particular because we do not really need to simulate a call stack). Moreover all the simulating code is deterministic but the code that simulates thread switching. We aim at the mechanization of this proof in the interactive theorem prover COQ. A non-trivial first step was to formalize the languages and their semantics, as well as the transformation. The next step will be to write the correctness proof itself with COQ.

The CONC2SEQ plugin does not only transform the code to verify. CONC2SEQ provides extensions to the ACSL behavioral specification language in order to write contracts for concurrent C programs. These assertions are also transformed by the plugin. Ultimately we would like to formalize axiomatic semantics for the parallel and sequential languages, and verify that the transformation of both code and assertions is such that a proof (using the sequential axiomatic semantics) of a simulating program allows to build a proof (using the parallel axiomatic semantics) of the source concurrent program. This is however a long term goal. Future work also includes extensions to the plugin itself that could also be verified as extensions of the current formal framework.

References

[1] June Andronick, Corey Lewis, Daniel Matichuk, Carroll Morgan & Christine Rizkallah (2016): *Proof of OS Scheduling Behavior in the Presence of Interrupt-Induced Concurrency*. In: *Interactive Theorem Proving - 7th International Conference, ITP 2016, Nancy, France, August 22-25, 2016, Proceedings*, pp. 52–68, doi:10.1007/978-3-319-43144-4_4.

[2] Patrick Baudin, Jean C. Filliâtre, Pascal Cuoq, Claude Marché, Benjamin Monate, Yannick Moy & Virgile Prevosto (2015): *ACSL: ANSI/ISO C Specification Language*. `http://frama-c.com/download.html`.

[3] Allan Blanchard, Nikolai Kosmatov, Matthieu Lemerre & Frédéric Loulergue (2015): *A Case Study on Formal Verification of the Anaxagoros Hypervisor Paging System with Frama-C*. In: *International Workshop on Formal Methods for Industrial Critical Systems (FMICS)*, LNCS, Springer, Oslo, Norway, pp. 15–30, doi:10.1007/978-3-319-19458-5_2.

[4] Allan Blanchard, Nikolai Kosmatov, Matthieu Lemerre & Frédéric Loulergue (2016): *Conc2Seq: A Frama-C Plugin for Verification of Parallel Compositions of C Programs*. In: *16th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM)*, IEEE, Raleigh, NC, USA, pp. 67–72, doi:10.1109/SCAM.2016.18.

[5] Allan Blanchard, Nikolai Kosmatov & Frédéric Loulergue (2017): *Concurrent Program Verification by Code Transformation: Correctness*. Research Report RR-2017-03, LIFO, Université d'Orléans. Available at `http://www.univ-orleans.fr/lifo/prodsci/rapports/RR/RR2017/RR-2017-03.pdf`.

[6] Bernd Fischer, Omar Inverso & Gennaro Parlato (2013): *CSeq: A concurrency pre-processor for sequential C verification tools*. In: *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pp. 710–713, doi:10.1109/ASE.2013.6693139.

[7] Jean Fortin (2013): *BSP-Why: a Tool for Deductive Verification of BSP Programs*. Ph.D. thesis, Université Paris-Est Créteil, LACL. Available at `http://hal.archives-ouvertes.fr/tel-00974977/`.

[8] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre & Gennaro Parlato (2014): *Bounded Model Checking of Multi-threaded C Programs via Lazy Sequentialization*. In: *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pp. 585–602, doi:10.1007/978-3-319-08867-9_39.

[9] Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles & Boris Yakobowski (2015): *Frama-C: A software analysis perspective*. *Formal Asp. Comput.* 27(3), pp. 573–609, doi:10.1007/s00165-014-0326-7.

[10] Nikolai Kosmatov, Virgile Prevosto & Julien Signoles (2013): *A Lesson on Proof of Programs with Frama-C. Invited Tutorial Paper*. In: *TAP*, *LNCS* 7942, Springer, pp. 168–177, doi:10.1007/978-3-642-38916-0_10.

[11] Akash Lal & Thomas W. Reps (2008): *Reducing Concurrent Analysis Under a Context Bound to Sequential Analysis*. In: *Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings*, pp. 37–51, doi:10.1007/978-3-540-70545-1_7.

[12] L. Lamport (1979): *How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Program*. *IEEE Trans. Comput.* 28(9), pp. 690–691, doi:10.1109/TC.1979.1675439.

[13] Xavier Leroy (2009): *A Formally Verified Compiler Back-end*. *Journal of Automated Reasoning* 43(4), pp. 363–446, doi:10.1007/s10817-009-9155-4.

[14] Truc L. Nguyen, Bernd Fischer, Salvatore La Torre & Gennaro Parlato (2016): *Lazy Sequentialization for the Safety Verification of Unbounded Concurrent Programs*. In: *Automated Technology for Verification and Analysis - 14th International Symposium, ATVA 2016, Chiba, Japan, October 17-20, 2016, Proceedings*, pp. 174–191, doi:10.1007/978-3-319-46520-3_12.

[15] T. Nipkow & L. Prensa Nieto (1999): *Owicki/Gries in Isabelle/HOL*. In J.-P. Finance, editor: *Fundamental Approaches to Software Engineering, Second International Conference (FASE'99)*, LNCS 1577, Springer, pp. 188–203, doi:10.1007/3-540-49020-3_13.

[16] S. Owicki & D. Gries (1976): *Verifying properties of parallel programs: an axiomatic approach*. *Communications of the ACM* 19(5), pp. 279–285, doi:10.1145/360051.360224.

[17] Shaz Qadeer & Dinghao Wu (2004): *KISS: keep it simple and sequential*. In: *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation 2004, Washington, DC, USA, June 9-11, 2004*, pp. 14–24, doi:10.1145/996841.996845.

[18] Ermenegildo Tomasco, Omar Inverso, Bernd Fischer, Salvatore La Torre & Gennaro Parlato (2015): *Verifying Concurrent Programs by Memory Unwinding*. In: *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pp. 551–565, doi:10.1007/978-3-662-46681-0_52.

[19] Ermenegildo Tomasco, Truc Nguyen Lam, Omar Inverso, Bernd Fischer, Salvatore La Torre & Gennaro Parlato (2016): *Lazy Sequentialization for TSO and PSO via Shared Memory Abstractions*. In: *Formal Methods in Computer-Aided Design (FMCAD)*, doi:10.1109/FMCAD.2016.7886679.