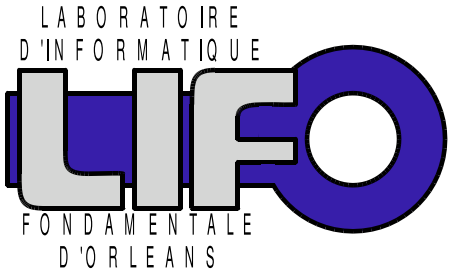

4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

Rapport de Recherche

Concurrent Program Verification by Code Transformation: Correctness

Allan Blanchard
LIFO, Université d'Orléans, France

Nikolai Kosmatov
CEA, LIST, France

Frédéric Loulergue
SICCS, Northern Arizona University, USA

Rapport n^o **RR-2017-03**

Concurrent Program Verification by Code Transformation: Correctness

Allan Blanchard
Univ. Orleans, INSA Centre Val-de-Loire
LIFO EA 4022, Orléans, France
Allan.Blanchard@univ-orleans.fr

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

Frédéric Loulergue
School of Informatics, Computing and Cyber Systems
Northern Arizona University
frederic.loulergue@nau.edu

February 2017

Abstract

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.

Contents

1 Introduction **4**

2 Considered Languages **5**
2.1 Syntax and Program Definition 5
2.2 Semantics 7
2.2.1 States 7
2.2.2 Operational semantics 8

3 Program Transformation **11**
3.1 Overview 11
3.2 Memory Allocation 12
3.3 Local and global assignments 14
3.4 Conditionals and Loops 15
3.5 Procedure call and return 16
3.6 Atomic block 18
3.7 Instruction and procedure transformation 18
3.8 Interleavings 18
3.9 Simulating program definition 20

4 Correctness **22**
4.1 Equivalence of states and traces 22
4.2 Correctness of the simulation 24
4.2.1 Initialization 25
4.2.2 Forward simulation 26
4.2.3 Backward simulation 27
4.2.4 Proof of the simulation theorem 28
4.2.5 Termination 28
4.3 Forward simulation of instructions 29
4.3.1 Local assignment 31
4.3.2 Memory load 32
4.3.3 Memory store 33
4.3.4 Conditional instruction 34
4.3.5 Loop 35
4.3.6 Call 36
4.3.7 Return 37
4.3.8 Atomic blocks 38

5 Towards a Mechanized Proof of Correctness **38**

6 Related Work **39**

7 Conclusion **41**

1 Introduction

Frama-C [14, 13] 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 [4]. 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 [6], we have proposed a new plugin, named conc2seq [7], 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 [16], 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 target 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. In a way, such a location is a kind of array.

The set of values that can be assigned to variables is written $\mathcal{V}$ 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 $\mathcal{X}$ 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 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:

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

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 (local variables) and the sequence of instructions that form its body:

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

where *Name* is the set of valid procedure names. **select**, **interleavings**, and names built from $\mathbb{Z}$ are all reserved names. $\mathcal{C}$ 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 $\{\mathit{instr}_1; \mathit{instr}_2\}$ instead of $\mathit{instr}_1; \mathit{instr}_2; \{\}$), conditionals, loops. Assignment is decomposed in three distinct use cases: assignment of a local variable with the value of an expression, writing the value of an expression to the heap, and reading of 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 that 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.

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

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 readings or procedure calls should be decomposed, and we support only the most structured control structures. The typing is also very basic: variables and heap accept any type of value (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 put into atomic sections. In the small imperative parallel language presented above, we could use the same technique: implement atomic operations as their sequential counterparts and put their calls into atomic blocks.

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. First we do not have names for global memory locations, but only constants for some statically allocated memory locations, and second the set of allocated memory locations does not depend on the number of threads.

The absence of global variable names is not an important limitation. 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

2.2.1 States

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

For both the sequential and the parallel languages, a heap $\eta : \mathbb{L} \rightharpoonup \mathbb{N} \rightharpoonup \mathcal{V}$ is a partial function from memory locations that returns a partial function from indices to values, thus

essentially defining an array indexed from 0. $\mathcal{H}$ 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 $\mathcal{L} = \mathit{Name} \times \mathcal{E} \times \mathcal{C}$. A call stack is defined as a sequence (stack) of local execution contexts: $s \in \mathcal{S} = \mathcal{L}^*$.

The states of sequential and parallel programs are respectively:

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

For a parallel state $\sigma_{par} \in \Sigma_{par}$, we denote by $\mathit{stacks}_{\sigma_{par}}$ 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 $[(\mathit{main},\ \varnothing,\ \mathit{body}(\mathit{main}))]$ for a sequential program. For a parallel program, the initial context of a thread $t \in \mathbb{T}$ is $[(\mathit{mains}(t),\ \varnothing,\ \mathit{body}(\mathit{mains}(t)))]$. For a sequential program, an initial state is thus: $([(\mathit{main},\ \varnothing,\ \mathit{body}(\mathit{main}))],\ \eta^{init}_{seq})$. For a parallel program, an initial state is $(\mathit{stacks}_{init},\ \eta^{init}_{par})$ where $\forall t \in \mathbb{T}.\ \mathit{stacks}_{init}(t) = [(\mathit{mains}(t),\ \varnothing,\ \mathit{body}(\mathit{mains}(t)))]$.

An initial heap η^{init}_{seq} should satisfy the memory allocation defined by a sequential program, i.e. if $(l, \mathit{size}) \in \mathit{mem}$ then $\eta^{init}_{seq}(l)(i)$ is defined for all $i < \mathit{size}$. In addition, the values contained at such a memory location cannot be themselves memory locations (but they can be any other value). The same constraints hold for a 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 $\exists \eta.\ \sigma^{final}_{par} = (\mathit{stacks},\ \eta)$ with $\forall t \in \mathbb{T}.\ \mathit{stacks}(t) = []$.

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. A program that does not terminate, is a safe program.

2.2.2 Operational semantics

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:

$$\begin{array}{rcl} a_{seq} & ::= & \tau \mid \textbf{call}\ m\ \overline{v} \mid \textbf{return}\ m \mid \textbf{read}\ l\ n\ v \mid \textbf{write}\ l\ n\ v \\ a_{par} & ::= & a_{seq} \mid \textbf{atomic}\ \overline{a_{seq}} \end{array}$$

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

$\mathcal{P} \vdash (m,\ \rho,\ (x := e;\ c)) \cdot s,\ \eta \xrightarrow{\tau} (m,\ \rho[x \mapsto v],\ c) \cdot s,\ \eta$ if $[\![e]\!]_\rho = v$ [**assign**]

$\mathcal{P} \vdash (m,\ \rho,\ (x[e_o] := e_v;\ c)) \cdot s,\ \eta \xrightarrow{\textbf{write } l\ o\ v} (m,\ \rho,\ c) \cdot s,\ \eta[(l,o) \mapsto v]$ if $[\![e_v]\!]_\rho = v,\ [\![e_o]\!]_\rho = o,\ \rho(x) = l,\ o < mem(l)$ [**read**]

$\mathcal{P} \vdash (m,\ \rho,\ (x := y[e_o];\ c)) \cdot s,\ \eta \xrightarrow{\textbf{read } l\ o\ v} (m,\ \rho[x \mapsto v],\ c) \cdot s,\ \eta$ if $[\![e_o]\!]_\rho = o,\ \rho(y) = l,\ o < mem(l),\ \eta(l)(o) = v$ [**write**]

$\mathcal{P} \vdash (m,\ \rho,\ (\textbf{while } e \textbf{ do } c_{body};\ c)) \cdot s,\ \eta \xrightarrow{\tau} (m,\ \rho,\ (c_{body} +\!\!+ \textbf{while } e \textbf{ do } c_{body};\ c)) \cdot s,\ \eta$ if $[\![e]\!]_\rho = \textbf{true}$ [**while : true**]

$\mathcal{P} \vdash (m,\ \rho,\ (\textbf{while } e \textbf{ do } c_{body};\ c)) \cdot s,\ \eta \xrightarrow{\tau} (m,\ \rho,\ c) \cdot s,\ \eta$ if $[\![e]\!]_\rho = \textbf{false}$ [**while : false**]

$\mathcal{P} \vdash (m,\ \rho,\ (\textbf{if } e \textbf{ then } c_t \textbf{ else } c_f;\ c)) \cdot s,\ \eta \xrightarrow{\tau} (m,\ \rho,\ (c_t +\!\!+ c)) \cdot s,\ \eta$ if $[\![e]\!]_\rho = \textbf{true}$ [**if : true**]

$\mathcal{P} \vdash (m,\ \rho,\ (\textbf{if } e \textbf{ then } c_t \textbf{ else } c_f;\ c)) \cdot s,\ \eta \xrightarrow{\tau} (m,\ \rho,\ (c_f +\!\!+ c)) \cdot s,\ \eta$ if $[\![e]\!]_\rho = \textbf{false}$ [**if : false**]

$\mathcal{P} \vdash (m,\ \rho,\ (m'(\overline{e});\ c)) \cdot s,\ \eta \xrightarrow{\textbf{call } m'\ \overline{v}} (m',\ [\overline{x} \mapsto \overline{v}],\ c_{m'}) \cdot (m,\ \rho,\ c) \cdot s,\ \eta$ if $m'(\overline{x})c_{m'} \in \mathcal{P},\ |\overline{x}| = |\overline{e}|,\ \overline{[\![e]\!]_\rho} = \overline{v},\ m' \notin s$ [**call**]

$\mathcal{P} \vdash (m,\ \rho,\ []) \cdot s,\ \eta \xrightarrow{\textbf{return } m} s,\ \eta$ [**return**]

$\mathcal{P} \vdash (m,\ \rho,\ (\textbf{select}_{\#tid}(tid,\ pc);\ c)) \cdot s,\ \eta \xrightarrow{\textbf{call select } [l_{tid},\ l_{pc}]} (m,\ \rho,\ c) \cdot s,\ \eta[(l_{tid}, 0) \mapsto t]$ if $[\![tid]\!]_\rho = l_{tid},\ [\![pc]\!]_\rho = l_{pc}$, and $0 \leq t < \#tid,\ \eta(l_{pc})(t) \neq 0$ [**select**]

Figure 1: Operational semantics of sequential programs

The operational semantics of sequential programs is defined in Figure 1. A judgement of the sequential semantics has the following form:

$$\mathcal{P} \vdash s,\ \eta \xrightarrow{a_{seq}} s',\ \eta'$$

meaning that a new state $(s',\ \eta')$ is reached from the state $(s,\ \eta)$ and this execution step produces an action a_{seq}. $\mathcal{P}$ is a program definition. We write

$$\mathcal{P} \vdash s,\ \eta \xrightarrow{\overline{a_{seq}}}{}^{*} s',\ \eta'$$

for the reflexive and transitive closure of the relation defined by the inference system of Figure 1.

We use the following notations: $l_1 +\!\!+ l_2$ is the concatenation of two sequences/lists. To add an element on top of a sequence, we use the symbol ; for sequences of instructions, and

$$\mathcal{P},\ \#tid \vdash stacks,\ \eta \xrightarrow{(t,a_{seq})} stacks[t \mapsto s'],\ \eta'$$
[**seq**] if $\mathcal{P} \vdash stacks(t),\ \eta \xrightarrow{a_{seq}} s',\ \eta'$ and $0 \leq t < \#tid$

$$\mathcal{P},\ \#tid \vdash stacks,\ \eta \xrightarrow{(t,\textbf{atomic } \overline{a_{seq}})} stacks[t \mapsto (m,\ \rho',\ c)],\ \eta'$$
[**atomic**] if $\mathcal{P} \vdash [(m,\ \rho,\ c_{atomic})],\ \eta \xrightarrow{\overline{a_{seq}}}{}^{*} [(m,\ \rho',\ [])],\ \eta'$
where $stacks(t) = (m,\ \rho,\ (\textbf{atomic}(c_{atomic});\ c)) \cdot s$ and $0 \leq t < \#tid$

Figure 2: Operational semantics of parallel programs

the symbol $\cdot$ 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 we abuse this notation in the case x is one component of a tuple in the list of tuples l. $f[a \mapsto b]$ is the function f' such that for all element a' different from a, then $f'(a') = f(a')$ and $f'(a) = b$. 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 $\eta[(l, o) \mapsto v]$ denote un 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.

$[\![e]\!]_\rho$ corresponds to the evaluation of expression e in local environment ρ. We omit the definition of this evaluation that is very usual, 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. 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 is procedure is not supposed to be called in parallel programs.

Figure 2 presents the semantics of parallel programs. A judgement of this semantics have the following form:

$$\mathcal{P},\ \#tid \vdash stacks,\ \eta \xrightarrow{(t,a_{par})} stacks',\ \eta'$$

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

Here, we present the transformation function that allows to get the sequential program that corresponds to the original parallel program.

3.1 Overview

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 `from`(m) of size $\#tid$. *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. `from`(m) is used to manage the return of calls to m in the simulating code.

All instructions are supposed to be atomic but loops and conditionals. For these, 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 this police are fresh variables not used in the input parallel program, but that we need to implement 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 tmp $:= \&y;\ y :=$ tmp[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.

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 $\mathit{begin}(m)$ for a call to m). Also for each procedure m we generate an additional procedure, named $\mathit{end}(m)$, 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:

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

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. It is essentially a loop that randomly selects a thread to execute and then switch to the appropriate procedure call depending on the program counter of the active thread.

3.2 Memory Allocation

For each local variable x possibly allocated by procedure inside its execution context, we create a simulating memory block at the address $\&x$, and that is defined for any correct thread identifier. These addresses are disjoint from each other and disjoint from the original addresses in the parallel program. Each of these new memory blocks contains, during the execution, for each thread, the current value of the corresponding original local variable in a parallel execution.

For each instruction, we create a procedure that simulate it. It receives in parameter the identifier of the thread that is simulated for this step of simulation in a local variable tid. We define the *load* function that associate, to a given local variable x of the original program, the sequence of simulation instructions that allows to load the corresponding value from the simulation block, for the thread identifier received in input of the simulating procedure. In the generated instruction, we reuse the original name of the local variable to create a new variable that is local to the simulating procedure:

$$load(x) \equiv \mathsf{tmp} := \&x;\ x := \mathsf{tmp}[\mathsf{tid}]$$

In this function, we use a local variable tmp. We suppose that this variable do not exists in the simulation procedure. We could reuse the name x since variable are not typed, we prefer to use tmp to keep in mind that it is only used to store simulation addresses and make the figures more readable.

The *variables* function returns the set of variables of its argument that can be an expression, an instruction, a bloc of instruction or even a program. We name *loads* the function that produces the instructions that loads all local variables from simulating addresses, iterating *load* on each of them. For example, to get all loads of the local variables corresponding to a given expression, we write:

$$loads(variables(e))$$

We define globally, in the simulating program heap:

- `ptid`, an address to a memory block of size 1, that is used to store the identifier of the currently executed thread,
- `pct`, an address that associates to each thread identifier its program counter,
- addresses for each procedure m, that associate for each thread identifier, the program point where it has to return after execution of m.

More precisely for this last item, we define the function `from` that returns, for a procedure m, the corresponding address. For example, if m is called by m_2, during the execution of the instruction c by the thread t, $\texttt{from}(m)[t]$ will contain the identifier of the instruction c according to the control flow graph.

We consider that the control flow graph is computed in advance for the original program and labels each instruction with a unique identifier $\ell \in \mathbb{Z}$. Each instruction is also labeled with the identifier of the instruction that follows it ℓ_{next} in the order of instructions. Notice that for example, for a list of instructions :

$$\textbf{if } e \textbf{ then } c_t \textbf{ else } c_f;\ c$$

the instruction that follows the conditional is the first instruction of c and not the instructions of the internal instruction blocks. In the rest of this paper, when we want to talk about label information, we will write the corresponding instruction $instr^{\ell}_{\ell_{next}}$.

```
1 trans_assign(x, e, ℓ, ℓ_next) ≡
2   toName(ℓ)(tid) {
3     loads(variables(e)) ;
4     tmp := &x ;
5     tmp[tid] := e ;
6     tmp := pct ;
7     tmp[tid] := ℓ_next ;
8   }
```

Figure 3: Assignment simulation

For ℓ_{next} we should consider the cases where there is not any instruction after a given instruction, either because this instruction is in conditional block or it is the last instruction of its procedure m. In the first case, if we are in a loop, ℓ_{next} is the identifier of the loop itself, if we are in the block of a conditional, it is the instruction that follows the conditional instruction. Finally, if it is the last instruction of a procedure m, the value of ℓ_{next} is a special label $end(m)$ that indicates we have to perform a return action. This label is distinct from all other labels in the program.

Indeed, the control flow graph also associates labels to each procedure: the identifier of its first instruction, and the one that corresponds to the return. This last identifier is not the identifier of an instruction in the source program but simply marks the end of the procedure, for which we have to produce a simulating procedure.

In the rest of this report, we suppose that the parallel semantics use labeled instructions.

3.3 Local and global assignments

The assignment $(\texttt{x := e})^{\ell}_{\ell_{next}}$ is simulated by a procedure named $toName(\ell)$ as illustrated in Figure 3. The first instructions of the simulation are generated using the *load* function previously explained in order to load the values of the local variables used in e for simulated thread. We then write the memory location &x that simulates x at the index tid, with the expression e for which local variables have been loaded by the instructions generated by the *load* function. Finally we place the program counter to the instruction ℓ_{next} for the thread tid.

The memory load $(\texttt{x := p[o]})^{\ell}_{\ell_{next}}$ is simulated by the procedure given in Figure 4. Following the same scheme that the one used in the local assignment, we first load local variables of o, and the local variable corresponding to the p pointer. Then, we load the global memory to x and write the read value to the memory block &x for the thread tid. Finally, we place the program counter on the next instruction.

The memory store instruction $(\texttt{p[o] := e})^{\ell}_{\ell_{next}}$ is translated as shown in Figure 5. We first load the local variables of e and o. If some variables are used in both expressions, it does not cause any problem from an execution point of view: such a variable will simply be read a second time, overwriting the previous read with the same value (since it cannot

```
1  trans_read(x, p, o, ℓ, ℓ_next) ≡
2    toName(ℓ)(tid) {
3      loads(variables(o)) ;
4      load(p) ;
5      x := p[o] ;
6      tmp := &x ;
7      tmp[tid] := x ;
8      tmp := pct ;
9      tmp[tid] := ℓ_next ;
10   }
```

Figure 4: Memory load simulation

```
1  trans_write(p, o, e, ℓ, ℓ_next) ≡
2    toName(ℓ)(tid) {
3      loads(variables(e)) ;
4      loads(variables(o)) ;
5      load(p) ;
6      p[o] := e ;
7      tmp := pct ;
8      tmp[tid] := ℓ_next ;
9    }
```

Figure 5: Memory store simulation

change between these two loads). We then load the pointer `p` and write the value in memory. Finally, we place the program counter on the next instruction.

3.4 Conditionals and Loops

A conditional instruction

$$(\textbf{if } e \textbf{ then } c_t \textbf{ else } c_f)^{\ell}_{\ell_{next}}$$

is simulated by the procedure of Figure 6. We first load the variables of `e`. We then build a new conditional instruction where branches now change the program counter according to the next instruction to execute in the corresponding branch. If this branch is empty, the program counter is the instruction that follows the conditional.

A loop instruction $(\textbf{while } e \textbf{ do } c)^{\ell}_{\ell_{next}}$ is simulated by a procedure generated as indicated in Figure 7. If the body of the loop is empty, the loop is infinite if the expression is evaluated to true: expressions cannot contain global memory loads, so the condition will always be true. The first branch of the generated conditional instruction move the program counter to the loop instruction itself if the body is empty, else, to the first instruction of the body. The

```
1  trans_cond(e, ct, cf, ℓ, ℓ_next) ≡
2    toName(ℓ)(tid) {
3      loads(variables(e)) ;
4      tmp := pct ;
5      if e then
6        ct = {} →
7          tmp[tid] := ℓ_next ;
8        ct = instr^ℓ'_ℓ'_next ; _ →
9          tmp[tid] := ℓ' ;
10     else
11       cf = [] →
12         tmp[tid] := ℓ_next ;
13       cf = instr^ℓ'_ℓ'_next ; _ →
14         tmp[tid] := ℓ' ;
15   }
```

Figure 6: Conditional instruction simulation

other branch move the program counter to the instruction that follows the loop.

3.5 Procedure call and return

Figure 8 gives the transformation of a procedure call $(\texttt{m(l)})^{\ell}_{\ell_{next}}$. As we previously mentioned, for each procedure, we create an address that associates, to each thread, the next instruction to perform when the current simulated procedure returns. This address is obtained using the `from` function.

First, we load all variables used in the list of parameters transmitted to the procedure `m`. We define the *combine* function that iterates on variables of `m` and write at its simulation address, for the current thread, the expression of `l` that it receives. We can define *combine*(, a, s) follows:

$$
\begin{array}{l}
combine(as,\ es,\ \mathsf{tid}) \equiv \\
\quad as, es \ = \ [], [] \ \rightarrow \ \{\} \\
\quad as, es \ = \ a :: as', e :: es' \ \rightarrow \ [\mathsf{tmp} := \&a;\ \mathsf{tmp}[\mathsf{tid}] := e] +\!\!+ combine(as',\ es',\ \mathsf{tid})
\end{array}
$$

Then, we write in $\texttt{from}(m)$, at the index tid, the identifier of the next instruction to execute at the return of the called procedure. Finally, we move the program counter to the first instruction of the body of `m` (note that if it is empty $\ell = \ell_{next}$).

When we process original procedure, we also add a simulation for the return that is identified ℓ_{next}. This simulation is defined as illustrated by the figure 9.

We load, from `from` (written during the simulation of the call to `m`), the identifier of the instruction where we have to return and move the program counter to this identifier. The `aux` local variable is introduced ensuring it is different of tmp and tid.

```
1  trans_loop(e, b, ℓ, ℓ_next) ≡
2    toName(ℓ)(tid) {
3      loads(variables(e)) ;
4      tmp := pct ;
5      if e then
6        b = {} → //infinite loop
7          tmp[tid] := ℓ ;
8        b = instr^{ℓ'}_{ℓ'_next} ; _ →
9          tmp[tid] := ℓ' ;
10     else
11       tmp[tid] := ℓ_next ;
12   }
```

Figure 7: Loop simulation

```
1  trans_call(m, begin(m), l, ℓ, ℓ_next) ≡
2    toName(ℓ)(tid) {
3      loads(variables(l)) ;
4      combine(args(m), l, tid) ;
5
6      tmp := from(m) ;
7      tmp[tid] := ℓ_next ;
8
9      tmp := pct;
10     tmp[tid] := ℓ_m ;
11   }
```

Figure 8: Procedure call simulation

```
1  trans_return(m, ℓ_end) ≡
2    toName(ℓ)(tid) {
3      tmp := from(m) ;
4      aux := tmp[tid];
5      tmp := pct;
6      tmp[tid] := aux ;
7    }
```

Figure 9: Procedure return simulation

Each thread receives its *main* procedure. Each of them are also processed to generated simultation procedures for their instructions. A parallel program is then a list of *main* procedure calls. The return identifier of each of them is 0, that is a reserved identifier and that corresponds to the action "do not perform action". We must also ensure that initially, every program counter is placed on the identifier of the call simulation of its *main*.

3.6 Atomic block

In order to produce the simulation of an atomic block, we can visit recursively the instructions to execute. Assignments are replaced by the simulation codes previously defined (we can remove the write of the program counter, but keeping it do not break the semantics). Loads needed for the expressions used in conditional instructions are produced the same way we perform them in the simulating procedure. However, the blocks now recursively contains the simulations of the instructions of the original code and not a jump to another program point. For the loop instruction, we must also reload local variables in the end of the block in order to update the expression of the condition.

Procedure call from an atomic block are simply inlined. We just have to build the execution context. Consequently, the idea is to build the context (as we do in the simulation of a call), and then insert the simulation of each of the instructions of the procedure. We also add the simulation of the return in order to simplify the trace equivalence that we will define in the section 4. Again, recursive calls are not allowed.

Finally, we can update the program counter to the instruction that follows the atomic block.

The figure 10 informally illustrates a transformation function for atomic blocks. We suppose that we have transformation functions for assignment, reads and writes that only return the code of the simulation, without the program counter update and the procedure itself (we note this variation `t_code` for the transformation `t`). Since recursive calls are not allowed, this function terminates.

3.7 Instruction and procedure transformation

To translate any instruction of the parallel language, we just pattern match on the instruction and one of the previously defined transformation functions is appropriately called. We define the function that calls the right transformation according to the type of instruction in Figure 11.

For each procedure in a program definition, this function is apply to each of the instruction of the procedure body.

3.8 Interleavings

The main procedure of the simulating sequential program is built as shown in Figure 12. In order to make the code more readable, we use a `switch` instruction that does not exists in the language. In the actual formalization, we chain conditional instructions.

```
1  trans_atomic_list(l) ≡
2    l = {} → {}
3    l = i :: l' →
4        sim :=
5              i = x := e     → trans_assign_code(x,e)
6              i = x := p[o]  → trans_read_code(x,p,o)
7              i = p[o] := e  → trans_write_code(p,o,e)
8              i = if e then ct else cf →
9                loads(variables({)e) ;
10               if e then trans_atomic_list(ct)
11               else      trans_atomic_list(cf)
12             i = while e do b →
13               loads(variables(e)) ;
14               while e do {
15                 trans_atomic_list(b) ;
16                 loads(variables({)e) ;
17               }
18             i = m(ps)^{ℓ'}_{ℓ_next}  →
19               {
20                 trans_call_code(m, ps) ;
21                 trans_atomic_list(body(m)) ;
22                 end(m)(tid)
23               }
24       sim ++ trans_atomic_list(l')
25
26 trans_atomic(l, ℓ, ℓ_next) ≡
27   toName(ℓ)(tid) {
28     trans_atomic_list(l)
29     ⫲ {
30       tmp := pct ;
31       tmp[tid] := ℓ_next
32     }
```

Figure 10: Atomic blocks simulation

```
1 trans_instruction(instr^ℓ_ℓnext) ≡
2   instr = x := e                  → trans_assign(x,e,ℓ,ℓnext)
3   instr = x := p[o]               → trans_read(x,p,o,ℓ,ℓnext)
4   instr = p[o] := e               → trans_write(p,o,e,ℓ,ℓnext)
5   instr = if e then ct else cf    → trans_cond(e,ct,cf,ℓ,ℓnext)
6   instr = while e do c            → trans_loop(e,c,ℓ,ℓnext)
7   instr = atomic c                → trans_atomic(c,ℓ,ℓnext)
8   instr = m(ps)                   → trans_call(m,end(m),ps,ℓ,ℓnext)
```

Figure 11: Simulation of an instruction

This procedure 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 $\texttt{from}(mains(t))$ to 0 to stop the execution when the main procedure ends. c_{init} also initializes the local variable 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 (implemented as nested conditionals) 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 $c_{termination}$).

3.9 Simulating program definition

For parallel program:

$$prog_{par} = \overline{proc}\ memory\ mains$$

the generated simulating program follows this scheme:

$$prog_{sim} = [\textbf{interleavings}] +\!\!+ \overline{proc_{sim}} \quad mem_{sim}$$

where list $\overline{proc_{sim}}$ contains the simulating procedures of each instruction and procedure return of the original program.

The list of addresses is $mem_{sim} = memory +\!\!+ mem^{sim}_{sim}$ where the list mem^{sim}_{sim} contains the pairs:

- $(\texttt{ptid},\ 1)$ used by **select**,
- $(\texttt{pct},\ \#tid)$ the program counters,
- for each procedure m of the original program, a pair $(\texttt{from}(m),\ \#tid)$ to store return program points,

```
1  interleavings(){
2    // c_init
3    tmp := pct ;
4    ∀t ∈ [0, #tid[,
5      tmp[t] := begin(mains(t)) ;
6    ∀t ∈ [0, #tid[,
7      tmp := from(mains(t)) ;
8      tmp[t] := 0 ;
9    terminated := false;
10
11   while ¬terminated do {
12     // c_select
13     select_{#tid}(ptid, pct) ;
14     // c_sim
15     tmp := ptid ;
16     tid := tmp[0] ;
17     tmp := pct ;
18     aux := tmp[tid] ;
19     switch aux is { ℓ : toName(ℓ)(tid) }
20     // c_termination
21     terminated := true ;
22     tmp := 0 ;
23     while tmp < #tid do {
24       if pct[tmp] ≠0
25       then { terminated := false }
26       else { } ;
27       tmp := tmp + 1 ;
28     }
29   }
30 }
```

Figure 12: Main procedure of the simulating sequential program

- for each local variable x of each procedure m of the original program, a pair $(\&_m x,\ \#tid)$, that models its simulating memory.

4 Correctness

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^{par}_{sim} that replicates the original program heap η_{par} and η^{sim}_{sim} the addresses that simulate the local variables of s_{par}. This second part also includes program counter address `pct`, the address that allow thread selection `ptid`, and the addresses used for procedure returns `from`(m). For all these addresses, the memory initially allocated must be greater or equal to the maximum thread identifier.

When we want to select the part of η^{sim}_{sim} that simulates the thread t, we use the syntax $\eta^{sim}_{sim}[t]$. It partially applies the function defined by η^{sim}_{sim}, restraining it to index t. So the function $\eta^{sim}_{sim}[t](l)$ is $\eta_{sim}(l,t)$.

We define state equivalence as follows:

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

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

$$\forall t \in \mathbb{T},\ ctx \in \mathcal{L},\ s \in \mathcal{S}.\ stacks(t) = ctx \cdot s \Longleftrightarrow \eta^{sim}_{sim}[t](\texttt{pct}) = \textsc{next}(ctx) \tag{3a}$$

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

$$\forall t \in \mathbb{T}.\ \textsc{wf\_stack}(stacks(t),\ \eta^{sim}_{sim}[t]) \tag{4}$$

$$s_{sim} = (\textbf{interleavings},\ \rho_{sim},\ \{\textbf{while}\ \neg\mathsf{terminated}\ \textbf{do}\ (c_{select} +\!\!+ c_{sim} +\!\!+ c_{termination})\}) \tag{5}$$
$$\wedge\ \rho_{sim}(\mathsf{terminated}) = \textbf{true} \Longleftrightarrow \forall t \in \mathbb{T}.\ \eta^{sim}_{sim}[t](\texttt{pct}) = 0$$

$$\sigma_{par} \sim \sigma_{sim}$$

with :

$$\textsc{next}(ctx) \equiv \begin{array}{ll} ctx = (\_, \_, instr^{\ell}_{\ell_{next}} :: \_) & \rightarrow \ell \\ ctx = (m, \_, []) & \rightarrow end(m) \end{array}$$

and

$$\frac{}{\textsc{wf\_stack}([],\ \eta^{sim}_{sim}[t])} \qquad \frac{\eta^{sim}_{sim}[t](\texttt{from}(m)) = 0}{\textsc{wf\_stack}((m, \_, \_) \cdot [],\ \eta^{sim}_{sim}[t])}$$

$$\frac{\eta^{sim}_{sim}[t](\texttt{from}(m)) = \textsc{next}(ctx) \qquad \textsc{wf\_stack}(ctx \cdot s',\ \eta^{sim}_{sim}[t])}{\textsc{wf\_stack}((m, \_, \_) \cdot ctx \cdot s',\ \eta^{sim}_{sim}[t])}$$

The premise (1) expresses the fact that the original heap should be a sub-part of the simulating heap.

Then (2), for any local variable x of the original program, the value we can find at the simulating memory location $\&x$ at the index t must be equal to the value of x in the stack of t in the original program. For the sake of readability, we write $\rho \in \mathit{stacks}(t)$ to directly select ρ in every ctx of $\mathit{stackst}$. This property is only an implication (and not an equivalence) since we do not reinitialize variables when we simulate the return of a procedure. As we consider safe programs, a new call to the procedure does not make read access to local variables before initializing it. So it does not breaks our notion of equivalence.

Our program counters must be correct. We use the NEXT function that, for a given local context ctx returns the identifier of the next instruction identifier ℓ of m, or $\mathit{end}(()m)$, if there is not any other instruction to execute. In (3a), for any thread, if its stack is not empty, the program counter of this thread $\eta^{\mathit{sim}}_{\mathit{sim}}[t](\texttt{pct})$ must be the identifier returned by NEXT. If it is empty (3b), the program counter must be 0.

The stack must be correctly modeled (5). We define the recursive predicate WF_STACK. For a given stack s (of a thread t), and the simulation of local data of t, $\eta^{\mathit{sim}}_{\mathit{sim}}[t]$, checks that $\texttt{from}(m)$ correctly models s. If the stack is empty, there are not anything to constrain in $\texttt{from}(m)$ since there is not any procedure return to perform as (3b) ensures that the program counter is 0, that forbids any action for t. If there is a unique context in the stack, the last return must allow us to bring the program counter to 0 in order to stop the execution. Finally, if there is more than one context, the top context $(m, \_, \_)$ must return to the next instruction of ctx, the next context, and the rest of the stack must also be correctly modeled. Again, this is an implication since we do not reinitialize $\texttt{from}(m)$ when we return. We will prove that it does not break the equivalence.

Finally, we define equivalent states for simulating program states such that the next action to perform is the evaluation of the condition of the interleaving loop. The simulation of the execution of an instruction is the complete execution of: loop condition evaluation and then the body of the loop (if needed). This is modeled by the part (5) of the equivalence.

The equivalence of traces is defined on filtered list of events generated by the semantics.

In the execution of the simulating program, we ignore τ-actions and memory operations performed in $\eta^{\mathit{sim}}_{\mathit{sim}}$. We ignore all call to and return from simulating procedures except for calls to **select**, calls $\texttt{call\_}\ell\texttt{\_}m\texttt{\_simulation}$ to the simulation of a call to m and calls $\texttt{return\_}\ell_{\mathit{end}}\texttt{\_}m\texttt{\_simulation}$ to the simulation of a return from m. While filtering events we must keep their order.

Finally, two traces t_{par} are t_{sim} equivalent when filtered, if replacing:

- all $(t, \textbf{call}\ m\ \_)$ with $\textbf{call select}\ [l_{\texttt{ptid}},\ l_{\texttt{pct}}];\ \textbf{call}\ \texttt{call\_}\ell\texttt{\_}m\texttt{\_simulation}\ t$;
- all $(t, \textbf{return}\ m)$ with $\textbf{call select}\ [l_{\texttt{ptid}},\ l_{\texttt{pct}}];\ \textbf{call}\ \texttt{return\_}\ell_{\mathit{end}}\texttt{\_}m\texttt{\_simulation}\ t$;
- all $(t, \textbf{read}\ l\ n\ v)$ with $\textbf{call select}\ [l_{\texttt{ptid}},\ l_{\texttt{pct}}];\ \textbf{read}\ l\ n\ v$;
- all $(t, \textbf{write}\ l\ n\ v)$ with $\textbf{call select}\ [l_{\texttt{ptid}},\ l_{\texttt{pct}}];\ \textbf{write}\ l\ n\ v$;
- all (t, τ) with $\textbf{call select}\ [l_{\texttt{ptid}},\ l_{\texttt{pct}}]$;
- all $(t, \textbf{atomic}\ (\overline{a_{\mathit{seq}}}))$ with $\textbf{call select}\ [l_{\texttt{ptid}},\ l_{\texttt{pct}}];\ \mathit{replace}(\overline{a_{\mathit{seq}}})$

in t_{par}, we obtain t_{sim}. And if by the opposite operation on t_{sim} we obtain t_{par}. The $replace(\overline{a_{seq}})$ operation consists in performing the same replacement operation in $\overline{a_{seq}}$, but without adding the **select** actions.

4.2 Correctness of the simulation

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

From σ_{sim}^{init}, we can reach, by executing the initialization sequence c_{init}, a sequential state σ_{sim}^{0} equivalent to σ_{par}^{init} (by the definition previously explained).

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

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

The different parts of this theorem will be later proved, we first give some intuitions about the proof. The proof of this theorem is based on two observations about the parallel semantics and its translation to the simulating program.

The first one is that all the semantics is completely deterministic except for the choice of the executed thread which is not an operation of the program. Equivalently, in the simulating program, the only operation that is not deterministic is the call to the **select** procedure that models the non-deterministic behavior of the parallel semantic rules.

The second observation is the fact that 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 corresponding simulating procedure, is also deterministic. So, for any sequential operation, the generated simulating code is deterministic. Now, if we prove a forward simulation for a transformation and the resulting code is deterministic, then we also proved the transformation respects a backward simulation [17].

The proof of the theorem is performed by induction on traces. A step of this induction is illustrated by the figure 13. For the forward simulation, the induction is on the instructions, for the backward simulation, on the number of interleaving loop executions.

The transition from σ_{sim} to $\sigma_{sim:t}$ corresponds to the evaluation of the loop condition, followed by the operation **select** noted c_{select}. It models the choice of a thread, if it exists one, performed by the parallel semantics.

The transition from σ_{par} to σ'_{par} is the execution of the instruction, for a given thread t chosen by the parallel semantics, in the sequential semantics. The transition between $\sigma_{sim:t}$ and $\sigma'_{sim:t}$ corresponds to the reduction of the sequence c_{sim} :

```
1 tid  := ptid[0];
2 tmp  := pct;
3 stmt := tmp[tid];
4
5 switch stmt is [
```

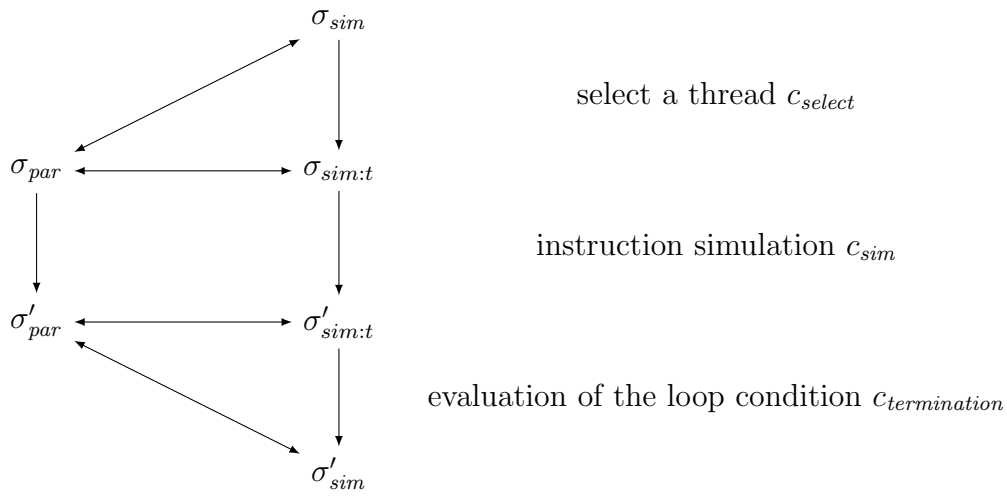

Figure 13: Simulation relation

```
∀m ∈ M, ∀instr^ℓ_ℓnext ∈ m,
ℓ : [ instr_stmt_typeℓ_simulation (tid) ]
]
```

The transition from $\sigma'_{sim:t}$ to σ'_{sim} corresponds to the reevaluation of the conditional of the loop, i.e. the evaluation of each program counter by the sequence of instructions $c_{termination}$.

4.2.1 Initialization

Lemma 2 (Initialization)**.** *Let $prog_{par}$ be a safe parallel program and $prog_{sim}$ its simulating program, σ^{init}_{par} an initial state of $prog_{par}$, and σ^{init}_{sim} an initial state of $prog_{sim}$. From σ^{init}_{sim}, we can reach, by the execution of the initialization sequence c_{init}, a sequential state σ^{0}_{sim} equivalent to σ^{init}_{par}.*

Proof. An initial state of the simulation is :

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

We also suppose that $\eta^{par}_{sim} = \eta_{par}$ and that η^{sim}_{sim} contains correctly allocated simulation blocks for local variables (cf. section 3.9). The part (1) of the equivalence is respected.

An initial state of $prog_{par}$ is: $(stacks_{init},\ \eta^{init}_{par})$ where

$$\forall t \in \mathbb{T}.\ stacks_{init}(t) = [(mains(t),\ \varnothing,\ body(mains(t)))]$$

The equivalence (2) is then respected.

The sequence c_{init} follows the following scheme.

```
tmp:=pct;
```

2 $\forall t \in [0, \#tid[,\ \texttt{tmp}[t]\texttt{:=}begin(mains(t))\texttt{;}$
3 $\forall t \in [0, \#tid[,\ \texttt{tmp:=from}(mains(t))\texttt{;}\ \texttt{tmp}[t]\ \texttt{:= 0;}$
4 $\texttt{terminated:=}\mathbf{false}\texttt{;}$

Applying the rule [**assign**] to get the address `pct` in `tmp`, and then the successive [**read**] rules for each program counter, we ensure that for η_{sim}^{sim}, (3a) is respected as identifiers of the first instructions of bodies are different of 0 if the body is not empty, and equal to 0 else, we have (3b).

Then, for each thread, we can place the pointer on $\texttt{from}(mains(t))$ (with the rule [**assign**] and then use [**read**] to ensure that the return program point is correctly placed and ensure that for η_{sim}^{sim}, (4) is verified.

We finally apply [**assign**] for the initialization of terminated. The program counters are different of 0 and the resulting state σ_{sim}^{0} is:

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

(5) is verified and our lemma is proved. □

4.2.2 Forward simulation

Lemma 3 (Forward simulation on a single step)**.** *Let $prog_{par}$ be a safe parallel program and $prog_{sim}$ its simulating program, σ_{par} and σ'_{par} two parallel states, σ_{sim} a state equivalent to σ_{par}, and (t, a_{par}) an action such that:*

$$prog_{par} \vdash \sigma_{par} \xrightarrow{(t,a_{par})} \sigma'_{par}$$

then there exists a trace tr such that trname is equivalent to $[(t, a_{par})]$ and:

$$prog_{sim} \vdash \sigma_{sim} \xrightarrow{tr}{}^{*} \sigma'_{sim}$$

and σ'_{sim} is equivalent to σ'_{par}.

Proof. By the equivalence relation, we know that *stsim* is of the form (omitting the name of the procedure):

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

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 $\eta_{sim}^{sim}[t](\texttt{pct}) \neq 0$, and consequently, by 5, $\rho_{sim}(\texttt{terminated}) =$ **false**. By [**while** : **true**] we get the simulating program state:

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

We then perform the reduction [**select**]. It generates an action **call select** $[l_{\texttt{ptid}},\ lpct]$ that places t at the address `ptid`, t being a allowed choice for *select* since $\eta_{sim}^{sim}[t](\texttt{pct}) \neq 0$. The resulting state is:

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

with $\eta'_{sim}(ptid, 0) = t$ and tr of the form $(\textbf{call select}\ [l_{\texttt{ptid}},\ l_{\texttt{pct}}]) :: trname'$.

At this step of the proof, we perform a case analysis depending on the executed instruction. We will explain this cases later. Let us admit, at the moment, the lemma 4.

Lemma 4 (Forward simulation of an instruction)**.** *Let i be an instruction and c_{sim} its simulating procedure. From:*

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

The execution of c_{sim} reaches a state:

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

such that (1), (2), (3a), (3b) *and* (4) *of the equivalence with σ'_{par} are verified, and the execution of c_{sim} produce a trace tr' that guarantees that $[(t, a_{par})]$, the action generated by i, is equivalent to* $(\textbf{call select}\ [l_{\texttt{ptid}},\ l_{\texttt{pct}}]) :: tr'$

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:

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

such that (5) is verified. Moreover, actions generated during this loop are reads in η^{sim}_{sim} and τ-actions (that are filtered).

We reach from a state σ_{sim} equivalent to σ_{par}, a new state σ'_{sim} equivalent to σ'_{par} with a trace tr equivalent to $[(t, a_{par})]$.

□

4.2.3 Backward simulation

Lemma 5 (Backward simulation on a single step)**.** *Let $prog_{sim}$ be the simulating program of a safe parallel program $prog_{par}$, σ_{sim} and σ'_{sim} two sequential states, σ_{par} a parallel state equivalent to σ_{sim}, $tr = (\textbf{call select}\ [l_{\texttt{ptid}},\ l_{\texttt{pct}}]) :: tr'$ a trace such that:*

$$prog_{sim} \vdash \sigma_{sim} \xrightarrow{tr}{}^{*} \sigma'_{sim}$$

and tr' does not contain call action to select, then it exists an action (t, a_{par}) such that $[(t, a_{par})]$ is equivalent to tr and;

$$prog_{par} \vdash \sigma_{par} \xrightarrow{(t,a_{par})} \sigma'_{par}$$

and σ'_{par} is equivalent to σ'_{sim}.

Proof. As $\sigma_{sim} \sim \sigma_{par}$, σ_{sim} is of the form:

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

The simulation builds a trace $tr = (\textbf{call select } [l_{\texttt{ptid}},\ l_{\texttt{pct}}]) :: tr'$ so the condition of the loop is evaluated to true (else we would not execute the loop, and the first action of the trace would not be realized). By the rule [**while** : **true**] the resulting state is:

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

and we know that it exists t such that $\eta^{sim}_{sim}(pct) \neq 0$. By (3a), we know that in the original program, it exists a thread t such that $stacks(t)$ is not empty.

In the simulating program, we then execute the thread selection instruction and we reach a state:

$$\big((\ldots,\ \rho_{sim},\ c_{sim} \# c_{termination} \#\{\textbf{while } \neg\mathsf{terminated} \textbf{ do } (c_{select} \# c_{sim} \# c_{termination})\}),\ \eta^1_{sim}\big)$$

such that $\eta^1_{sim}(ptid, 0) = t$, moreover by (3b), we know the identifier ℓ of the instruction to execute, and we know that in the original program, the next instruction to be performed by t is $instr^{\ell}_{\ell_{next}}$.

As $prog_{par}$ is safe, it does not block, the instruction $instr^{\ell}_{\ell_{next}}$ of t can be executed, and there exists a new parallel state σ'_{par}, reached with an action (t, a_{par}). By lemma 3, 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 $\textbf{call select } [l_{\texttt{ptid}}, l_{\texttt{pct}}]$ equivalent to the one produced for tr and represents the execution of $instr^{\ell}_{\ell_{next}}$ 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 $[(t, a_{par})]$. □

4.2.4 Proof of the simulation theorem

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

(1) From σ^{init}_{sim}, we can reach, by executing the initialization sequence c_{init}, a sequential state σ^0_{sim} equivalent to σ^{init}_{par} (by the definition previously explained).

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

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

Proof. (1) is proved using lemma 2.

(2) is proved by induction on the list of instruction to execute using the lemma 3.

(3) is proved by induction on the number of iteration of the interleaving loop using the lemma 5. □

4.2.5 Termination

Lemma 6 (Termination (forward simulation))**.** *Let $prog_{par}$ be a safe parallel program and $prog_{sim}$ its simulating program, σ^{init}_{par} an initial state of $prog_{par}$ and σ^{init}_{sim} an initial state of the simulation.*

Form σ_{sim}^{init} can reach, by executing the initialization sequence c_{init} a sequential state σ_{sim}^{0} equivalent to σ_{par}^{init}.

For all state σ_{par}^{final} reachable from σ_{par}^{init}, there exists a state σ_{sim} equivalent to σ_{par}^{final}, reachable from σ_{sim}^{0} such that from σ_{sim}, we reach a final state σ_{sim}^{final}.

Proof. By the lemma 2, we can reach σ_{sim}^{0}.

By the theorem 1, we can reach, in the simulation, a state σ_{sim} equivalent to σ_{par}^{final}. In this case, according to (3b):

$$\forall t \in \mathbb{T}. \quad \eta_{sim}^{sim}[t](\mathtt{pct}) = 0$$

and by (5), $\rho_{sim}(\mathsf{terminated}) = vfalse$, so the state is:

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

is reduced (using [**while** : **false**]) in:

$$((\ldots,\ \rho_{sim},\ \{\}),\ \eta_{sim})$$

and then (using [**return**]):

$$([],\ \eta_{sim})$$

which is a final state. □

4.3 Forward simulation of instructions

In this section we the lemma 4 for each type of instruction. We come back to the context of the proof of the forward simulation, and we want to prove that for an action (t, a_{par}) of an instruction i that brings the program from σ_{par} to σ'_{par}, we can reach, from a state σ_{sim} equivalent to σ_{par}, by the execution of c_{sim} of i, with a trace tr equivalent to $[(t, a_{par})]$, a new state σ'_{sim} equivalent to σ'_{par}.

After the beginning of the proof (evaluation of the loop condition and execution of thread selection), we are in a simulation state:

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

where $\eta'_{sim}(ptid, 0) = t$, with a generated trace [**call select** $[l_{\mathtt{ptid}},\ l_{\mathtt{pct}}]$] and we supposed that we could, by the execution of b_{sim}, reach a state:

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

with a trace tr' such that: $tr =$ (**call select** $[l_{\mathtt{ptid}},\ l_{\mathtt{pct}}]$) $::$ tr' equivalent to $[(t, a_{par})]$, and this state respects the equivalence with σ'_{par} for the parts (1), (2), (3a), (3b) and (4) ((5) being ensured by the execution of the end of the loop body $c_{termination}$).

We now want to prove that c_{sim} respects our assumption for any atomic instruction to execute.

c_{sim} is of the form :

```
1 select_{#tid}(ptid, pct);
2 tmp:=ptid; tid:=tmp[0]; tmp:=pct; aux:=tmp[tid];
3 switch aux is { ℓ : toName(ℓ)(tid) }
```

For any instruction, the execution of the load of the thread identifier and the resolution of the program counter, and the call to the corresponding simulating procedure, proceeds in a similar way. We unfold this part of the proof before considering each specific instruction.

Before executing c_{sim}, the state is:

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

In a first time, we load the value at the memory location `ptid` in tid ([**write**] rule). Then, we load the value a the memory location `pct` for the thread tid ([**write**] rule). Finally we solve this program counter to get the identifier of the procedure to call by successive conditional instructions ([**if** : **false**] rule) until we reach the right one ([**if** : **true**] rule), adding at the head of the instruction list the call to the considered instruction, which brings us to the state:

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

where $\rho'_{sim} = \rho_{sim}[\mathsf{tid} \mapsto t][\mathsf{aux} \mapsto id]$.

The call of the procedure produces an action **call** $toName(\ell)\ t$, and a new context on the top of the stack (we ignore the tail of the stack in this formulation):

$$((toName(\ell),\ \varnothing[\mathsf{tid} \mapsto t],\ body(toName(\ell))) \cdot \ldots,\ \eta'_{sim})$$

During all these reductions, events are τ-actions and reads from the part η^{sim}_{sim} of ρ_{sim}, that are filtered.

By now, we show equivalence for each instruction, reaching the state:

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

in the simulation.

In the rest of this section, we name:

- η_{par}, the heap of the original program before the execution of the instruction,
- η_{par}, the heap of the original program after the execution of the instruction,
- $\rho_{par:t}$, the local environment of t before the instruction,
- $\rho_{par:t}$, the local environment of t after the instruction.

4.3.1 Local assignment

We recall the rule of assignment:

$$\mathcal{P} \vdash (m,\ \rho,\ (x := e;\ c)) \cdot s,\ \eta \xrightarrow[\text{if } [\![e]\!]_\rho = v]{\tau} (m,\ \rho[x \mapsto v],\ c) \cdot s,\ \eta$$
[**assign**]

In this instruction, the expression is composed of local variable and of constant values. We note x_1 to x_n the variables that compose the expression (that are not x itself, if it is part of the expression), and v_1 to v_n their values (if the step is performed, e can be evaluated so these values are defined).

Before the simulation operation, our assumption is that the state is correctly simulated. So, for any variable x_i of value v_i in the original program, $\eta^{sim}_{sim}[t](\&x_i) = v_i$.

Our state is:

$$((toName(\ell),\ \rho_{sim},\ body(toName(\ell))) \cdot \ldots,\ \eta'_{sim})$$

with $\rho_{sim} = \varnothing[tid \mapsto t]$.

And the body of the procedure contains the instructions of the simulation, so successive pairs (a):

```
1 tmp := &x_i ;
2 x_i := tmp[tid] ;
```

for each variable of e, followed by the write of x (b):

```
1 tmp := &x ;
2 tmp[tid] := e ;
```

and finally the move of the program counter to the next instruction:

```
1 tmp := pct ;
2 tmp[tid] := ℓ_next ;
```

After the execution of (a), we have updated ρ_{sim} such that $\forall x \in e.\rho_{sim}(x) = v_i$ since we write each in x_i, the value v_i that we find in the memory according to the simulation ($\eta^{sim\prime}_{sim}(\&x_i, t) = v_i$).

We then perform (b), the write of the new value of x at its simulating memory location. Since $\forall x \in e.\rho_{sim}(x) = v_i$, we have:

$$[\![e]\!]_{\rho'_{sim}} = [\![e]\!]_{\rho_{par:t}} = v_e$$

Consequently, if after the execution of the original code, we have $\rho'_{par:t}(x) = v$, we have in the simulation $\eta^{sim\prime\prime}_{sim}(\&x, t) = v$ that ensures that we maintain the part (2) of the equivalence.

Finally (c), the program counter is updated to the identifier of the next instruction. After the execution of the assignment, the parallel program state reduced to t is:

$$\eta_{par}, ((m, \rho[x \mapsto v], c) \cdot s)$$

with $\eta_{par} = \eta'_{par}$.

In the simulating program, the program counter now points to the simulation procedure of the first instruction of c, if it exists, else on the return simulation of m, that maintains the part (3a) of the relation. In the original code, the stack is not emptied by the instruction (even if it was the last instruction, since we have to execute the return operation), the identifier is different of 0, ensuring (3b).

The part (1) of the relation is maintained since the execution of the original program does not modify η_{par}, and the simulating program does not modify $\eta_{sim}^{par\prime}$. So $\eta_{par} = \eta'_{par}$ and equivalently $\eta_{sim}^{par\prime} = \eta_{sim}^{par\prime\prime}$.

The relation (4) is maintained since the execution of the original program do not modify the call stack, and our simulation code do not modify $\texttt{from}(m)$.

It brings us to a new state:

$$((toName(\ell),\ \rho'_{sim},\ \{\}) \cdot \ldots,\ \eta''_{sim})$$

and then by the procedure return, to:

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

In the execution of the original program, a τ-action is generated, So $(t, a_{par}) = (t, \tau)$. $tr = (\textbf{call select } [l_{\texttt{ptid}},\ l_{\texttt{pct}}]) :: tr'$, and $tr' = []$ since all actions generated by c_{sim} are in $\eta_{sim}^{sim\prime}$ and these actions are filtered. The traces are equivalent.

4.3.2 Memory load

We recall the rule of memory load (sightly adapted for readability):

$$\begin{array}{ll} \mathcal{P} \vdash (m,\ \rho,\ (x := p[o];\ c)) \cdot s,\ \eta & \xrightarrow{\textbf{read } l\ n\ v} \quad (m,\ \rho[x \mapsto v],\ c) \cdot s,\ \eta \\ [\textbf{read}] & \text{if } [\![o]\!]_\rho = n,\ \rho(p) = l,\ n < mem(l),\ \eta(l)(n) = v \end{array}$$

We recall that after simulating procedure call, the simulating state is:

$$((toName(\ell),\ \rho_{sim},\ body(toName(\ell))) \cdot \ldots,\ \eta'_{sim})$$

with $\rho_{sim} = \varnothing[tid \mapsto t]$. We do not explain the proof of the parts (3a), (3b) and (4) of the equivalence since it is exactly the same proof that we have performed for the assignment.

The list of instructions contains:

```
1 loads(variables(o)) ;
2 load(p) ;
3 x := p[o] ;
4 tmp := &x ;
5 tmp[tid] := x ;
```

The load of variables of `o` and of the local variable `p` is equivalent to the one used in the assignment. After these instructions, for each x_i of o, $\rho'_{sim}(x_i) = v_i$ if $\rho_{par:t}(x_i) = v_i$ and equivalently of the pointer `p` : $\rho'_{sim}(p) = l$ if $\rho_{par:t}(p) = l$.

Exactly like we had for e in the assignment, we have:

$$[\![o]\!]_{\rho'_{sim}} = [\![o]\!]_{\rho_{par:t}} = n$$

After the execution of the read in the original code, the heap η_{par} is not modified, and in the simulation $\eta^{par\prime}_{sim}$ is not modified. We only access to the address $\&x$ that is in $\eta^{sim\prime}_{sim}$, which is disjoint from $\eta^{par\prime}_{sim}$. Equivalence (1) is maintained.

By

$$[\![o]\!]_{\rho'_{sim}} = [\![o]\!]_{\rho_{par:t}} = n$$

and $\rho'_{sim}(p) = \rho_{par:t}(p) = l$, the instruction `x := p[o]` produce an action **read** $l\ n\ v$ equivalent to the one produced by the original program since (1) is maintained, $\eta_{par}(l, n) = v$ and $\eta_{sim}(l, n)' = v$, l being in $\eta^{par\prime}_{sim}$.

Consequently, the write performed at $\&x$ for the index tid of the value x restores the part (2) of the state equivalence since $\rho'_{par:t}(x) = \eta^{sim\prime\prime}_{sim}[t](x)$.

As mentioned, the proof of (3a), (3b) and (4) is equivalent to the one explained in the assignment.

In the execution of the original program, a read action is generated. So $(t, a_{par}) = (t, \textbf{read}\ l\ n\ v)$. $tr = \textbf{call select}\ [l_{\texttt{ptid}},\ l_{\texttt{pct}}] :: tr'$ and $tr' = [\textbf{read}\ l\ n\ v]$ since all other actions are filtered, which ensures the trace equivalence.

4.3.3 Memory store

We recall the rule of memory store (sightly adapted for readability):

$$\begin{array}{ll} \mathcal{P} \vdash (m,\ \rho,\ (p[o] := e;\ c)) \cdot s,\ \eta & \xrightarrow{\textbf{write}\ l\ n\ v} \quad (m,\ \rho,\ c) \cdot s,\ \eta[(l, o) \mapsto v] \\ [\textbf{write}] & \text{if } [\![e]\!]_\rho = v,\ [\![o]\!]_\rho = n,\ \rho(p) = l,\ n < mem(l) \end{array}$$

We recall that after simulating procedure call, the simulating state is:

$$((toName(\ell),\ \rho_{sim},\ body(toName(\ell))) \cdot \ldots,\ \eta'_{sim})$$

with $\rho_{sim} = \varnothing[tid \mapsto t]$. We do not explain the proof of the parts (3a), (3b) and (4) of the equivalence since it is exactly the same proof that we have performed for the assignment.

```
1 loads(variables(e)) ;
2 loads(variables(o)) ;
3 load(p) ;
4 p[o] := e ;
```

For each local variables x of e, o and p, we load it new local variables x in the simulating procedure. Consequently after these loads, we have:

- $\forall x \in variables(e).\rho'_{sim}(x) = v$ with $\rho_{par:t}(x) = v$;
- $\forall x \in variables(o).\rho'_{sim}(x) = v$ with $\rho_{par:t}(x) = v$;
- $\rho'_{sim}(p) = l$ avec $\rho_{par:t}(p) = l$

So, $[\![e]\!]_{\rho'_{sim}} = [\![e]\!]_{\rho_{par:t}} = v$, $[\![o]\!]_{\rho'_{sim}} = [\![o]\!]_{\rho_{par:t}} = n$ and $[\![p]\!]_{\rho'_{sim}} = [\![p]\!]_{\rho_{par:t}} = l$. The operation `p[o] := e` in the simulating code produce the action same **write** l n v produced by the original code execution. This operation is not filtered in the original program, nor in the simulating one, since it is performed in $\eta^{par\prime}_{sim}$. The trace equivalence is maintained.

Moreover, the only update in the global state with this simulation phase is in $\eta^{par\prime}_{sim}$ for the simulating program and in η_{par} for the original one. This update is equivalent (same action on the memory). (1) is maintained.

The local environment of t is not modified in the original program nor $\eta^{sim\prime}_{sim}$ in the simulating one, (2) is maintained.

In the execution of the original program, a write action is generated, so $(t, a_{par}) = (t, \textbf{write}\ l\ n\ v)$. $tr = \textbf{call select}\ [l_{\texttt{ptid}},\ l_{\texttt{pct}}] :: tr'$ and $tr' = [\textbf{write}\ l\ n\ v]$ since all other actions are filtered. The trace equivalence is maintained.

4.3.4 Conditional instruction

We recall the rule of conditional:

$$\mathcal{P} \vdash (m,\ \rho,\ (\textbf{if}\ e\ \textbf{then}\ c_t\ \textbf{else}\ c_f;\ c)) \cdot s,\ \eta \xrightarrow{\tau} (m,\ \rho,\ (c_t +\!\!+\, c)) \cdot s,\ \eta \quad \text{if } [\![e]\!]_\rho = \textbf{true} \quad [\textbf{if}{:}\textbf{true}]$$

$$\mathcal{P} \vdash (m,\ \rho,\ (\textbf{if}\ e\ \textbf{then}\ c_t\ \textbf{else}\ c_f;\ c)) \cdot s,\ \eta \xrightarrow{\tau} (m,\ \rho,\ (c_f +\!\!+\, c)) \cdot s,\ \eta \quad \text{if } [\![e]\!]_\rho = \textbf{false} \quad [\textbf{if}{:}\textbf{false}]$$

We recall that after simulating procedure call, the simulating state is:

$$((toName(\ell),\ \rho_{sim},\ body(toName(\ell))) \cdot \ldots,\ \eta'_{sim})$$

with $\rho_{sim} = \varnothing[tid \mapsto t]$. The list of instructions contains (according to `b1` and `b2` in the original code) the instructions:

```
1  loads(variables(e)) ;
2  tmp := pct ;
3  if e then
4    ct = {} →
5      tmp[tid] := ℓ_next ;
6    ct = instr^ℓ'_ℓ'_next ; _ →
7      tmp[tid] := ℓ' ;
8  else
9    cf = [] →
10     tmp[tid] := ℓ_next ;
11   cf = instr^ℓ'_ℓ'_next ; _ →
12     tmp[tid] := ℓ' ;
```

In a first time, we load the local variables of e from $\eta^{sim\prime}_{sim}$. Equivalently to the assignment, we know that:

$$[\![e]\!]_{\rho_{sim}} = [\![e]\!]_{\rho_{par:t}} = v$$

As these evaluation are equivalent, the executions of the original program and the simulating one follow the same branches for equivalent environments.

The obtained state after this evaluation in the original program is

- $\eta_{par}, ((m, \rho, (\texttt{ct} +\!\!+ c)) \cdot s)$ or
- $\eta_{par}, ((m, \rho, (\texttt{cf} +\!\!+ c)) \cdot s)$

For each branch, either the body is empty or it is not. So either the new state is $c_{chosen} +\!\!+ c$ or simply c, c being possibly empty.

If the chosen bloc is not empty, the program counter now points to the first instruction of the block, else, it is moved to the first instruction of c (that we assume to be correctly computed by the control flow graph). We reach the first instruction to execute, or the end of the procedure ($end(m)$). We ensure (3a), and (3b) as we do not pop context from the stack.

η_{par} and $\eta_{sim}^{par\prime}$ are not modified, so (1) is maintained. We do not pop a context in the original program, and we do not write $\texttt{from}(\_)$, so (4) is maintained. Finally, we do not modify local variables in the original program and we do not modify η_{sim}^{sim} in the simulation, so (2) is also maintained.

In the original program, a τ-action is generated, so $(t, a_{par}) = (t, \tau)$. $tr = (\textbf{call select}\ [l_{\texttt{ptid}},\ l_{\texttt{pct}}]) :: tr'$, and $tr' = []$ since all actions of c_{sim} are filtered. We ensure the traces equivalence.

4.3.5 Loop

We recall the rule of loop:

$$\mathcal{P} \vdash (m,\ \rho,\ (\textbf{while}\ e\ \textbf{do}\ c_{body};\ c)) \cdot s,\ \eta \xrightarrow{\tau} (m,\ \rho,\ (c_{body} +\!\!+ \textbf{while}\ e\ \textbf{do}\ c_{body};\ c)) \cdot s,\ \eta \quad \text{if } [\![e]\!]_\rho = \textbf{true} \quad [\textbf{while:true}]$$

$$\mathcal{P} \vdash (m,\ \rho,\ (\textbf{while}\ e\ \textbf{do}\ c_{body};\ c)) \cdot s,\ \eta \xrightarrow{\tau} (m,\ \rho,\ c) \cdot s,\ \eta \quad \text{if } [\![e]\!]_\rho = \textbf{false} \quad [\textbf{while:false}]$$

We recall that after simulating procedure call, the simulating state is:

$$((toName(\ell),\ \rho_{sim},\ body(toName(\ell))) \cdot \ldots,\ \eta'_{sim})$$

with $\rho_{sim} = \varnothing[tid \mapsto t]$. The list of instructions contains:

```
1 loads(variables(e)) ;
2 tmp := pct ;
3 if e then
4   b = {} → //infinite loop
5     tmp[tid] := ℓ ;
6   b = instr^{ℓ'}_{ℓ'_next} ; _ →
7     tmp[tid] := ℓ' ;
8 else
9   tmp[tid] := ℓ_next ;
```

The proof is similar to the proof of the conditional instruction. However, the validity of the simulation, from the execution order point of view and the trace equivalent is conditioned by the correct computation of the control flow graph and the fact that the last instruction of the loop body brings back to the simulation procedure of the loop.

4.3.6 Call

We recall the rule of procedure call:

$$\begin{array}{ll} \mathcal{P} \vdash (m,\ \rho,\ (m'(\overline{e});\ c)) \cdot s,\ \eta & \xrightarrow{\textbf{call}\ m'\ \overline{v}} \quad (m',\ [\overline{x} \mapsto \overline{v}],\ c_{m'}) \cdot (m,\ \rho,\ c) \cdot s,\ \eta \\ [\textbf{call}] & \text{if } m'(\overline{x})c_{m'} \in \mathcal{P},\ |\overline{x}| = |\overline{e}|,\ \overline{[\![e]\!]_\rho} = \overline{v},\ m' \notin s \end{array}$$

We recall that after simulating procedure call, the simulating state is:

$$((toName(\ell),\ \rho_{sim},\ body(toName(\ell))) \cdot \ldots,\ \eta'_{sim})$$

with $\rho_{sim} = \varnothing[tid \mapsto t]$. The list of instructions contains:

```
1 loads(variables(l)) ;
2 combine(args(m), l, tid) ;
3 tmp := from(m) ;
4 tmp[tid] := ℓ_next ;
5 tmp := pct;
6 tmp[tid] := ℓ_m ;
```

We do not modify η_{par} in the execution of the original code, and we do not modify $\eta^{par\prime}_{sim}$ in the execution of the simulating program. The relation (1) is maintained. With the same reasoning we had for the assignment, we know that after the loads of all local variables, we have: $\forall e \in l.[\![e]\!]_{\rho'_{sim}} = [\![e]\!]_{\rho_{par:t}} = v$. For each simulation address $\&v_i$ for the parameters of m_2, we write the value of the evaluation of e_i. The *combine* function produces a list of instructions:

```
1 tmp = &args(m)[i];
2 tmp[tid] = e_i;
```

When we call a procedure, the execution of the original program builds a new local environment in which each parameter x_i receives its argument of value $[\![e_i]\!]_{\rho_{par:t}}$. The generated instructions will ensure that for each simulating address $\&x_i$, for the thread t, we write $\eta^{sim\prime\prime}_{sim}(\&x_i) = [\![e_i]\!]_{\rho_{par:t}}$. Since, $\forall e \in l.[\![e]\!]_{\rho'_{sim}} = [\![e]\!]_{\rho_{par:t}} = v$, we maintain the equivalence (2) when modeling the local context of m_2.

Then, we write the identifier ℓ_{next} of the next instruction to execute in $\texttt{from}(m_2)$ for the thread t. A this point, we know that the stack of t contains at least an execution context (the one we are using). If this context is the only one, by (4), we know that $\texttt{from}(m_1) = 0$, we have to prove (4) for this call. Now, after the execution of the line 3, $\eta_{sim}{}''[t](\texttt{from}(m_2))$ is equal to the identifier of the instruction that follows the call. Assuming that there is no recursive call, and that procedure return simulations also maintain (4), by induction the stack is correctly modeled.

Finally, we move the program counter to the first instruction of m_2, and its entire body is pushed on the stack. We maintain (3a) and (3b).

In the execution of the original program an action **call** m_2 _ is generated, so $(t, a_{par}) = (t, \textbf{call } m_2 \_)$. $tr = (\textbf{call select } [l_{\texttt{ptid}},\ l_{\texttt{pct}}]) :: tr'$ and $tr' = [\textbf{call } \texttt{call\_}\ell\texttt{\_}m_2\texttt{\_simulation } t]$ since all other actions are filtered. We also ensure traces equivalence.

4.3.7 Return

We recall the rule of procedure return:

$$\mathcal{P} \vdash (m,\ \rho,\ []) \cdot s,\ \eta \quad \xrightarrow{\textbf{return } m} \quad s,\ \eta$$

We recall that after simulating procedure call, the simulating state is:

$$((toName(\ell),\ \rho_{sim},\ body(toName(\ell))) \cdot \ldots,\ \eta'_{sim})$$

with $\rho_{sim} = \varnothing[tid \mapsto t]$. The list of instructions contains:

```
1 end(m)(tid) {
2   tmp := from(m);
3   aux := tmp[tid];
4   tmp := pct;
5   tmp[tid] := aux ;
6 }
```

We do not modify η_{par} in the execution of the original code, and we do not modify $\eta_{sim}^{par\prime}$ in the execution of the simulating program. The relation (1) is maintained.

Local variables are not modified in the execution of the original program and the simulation do not perform write actions in $\eta_{sim}^{sim\prime}$ for the simulating address of the local variables. (2) is maintained.

But, an execution context is popped, and the local variables of this context are not defined anymore. The values in $\eta_{sim}^{par\prime}$ for these local variables are however still defined, this is why we only have an implication. To get back an equivalence, we could have a mean to reinitialize value in the heap to be undefined. As we only consider safe programs, we have the guarantee that local variables are, in the original program, initialized before being read. It guarantees that the simulation is still correct, since if the procedure is called again, we will write $\eta_{sim}^{par\prime}$ before reading it again.

When we pop the execution context, the previous context is know at the top of the stack, if this context m' exists. By (4), we know that $\texttt{from}(m)$ for the current thread contains the next instruction of m' to execute. After popping, it is this context that is at the top of the stack and so the write of the program counter to this value guarantees the part (3a) of the equivalence.

If the local state only contains one context, by (4), we know that $\texttt{from}(m)$ is 0 and the program counter is moved to this identifier, guaranteeing that this thread will not be activated anymore. (3b) is maintained.

The implication (4) is maintained since we only pop the first context, so the tail of the stack is still correctly modeled. If we call m again later, (4) is still maintained since the call will overwrite $\texttt{from}(m)$.

In the execution of the original program an action **return** $m\_$ is generated, so $(t, a_{par}) = (t, \textbf{return}\ m\_)$. $tr = (\textbf{call select}\ [l_{\texttt{ptid}},\ l_{\texttt{pct}}]) :: tr'$ and $tr' = [\textbf{call}\ \texttt{return\_}\ell_{end}\texttt{\_}m\texttt{\_simulation}\ t]$ since all other actions are filtered. We also ensure traces equivalence.

4.3.8 Atomic blocks

Atomic blocks use the ideas of the preceding proofs. Instead of executing a unique step of simulation, we execute all simulation steps of all instructions of code block.

The equivalence for assignment, read and write is identical to what we already proved. For conditional instructions, the evaluation is equivalent, however, the execution of the chosen body is verified by induction on the correction execution of the translation of the atomic bloc. The idea is the same for the loop, but we also have to verify again the correct evaluation of the local variables at the end of the loop body.

Procedure calls first build the call context, then simulate the instruction and finally simulate the return. It generates the same state updates that the ones mentioned in previous proofs. By using the simulation procedure for call and return we ensure that we generate the same actions for the traces equivalence.

5 Towards a Mechanized Proof of Correctness

We aim at mechanizing the proof of correctness using the proof assistant Coq [23, 5]. A first step to do so is to formalize both languages and their semantics and to formalize the transformation. The current state of the development[1] includes this first step. It consists of a bit more than 3,000 lines of Coq, about 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 (an about 50% of it comes from another project with only slight modifications), and the remaining 30% is about the formalization of the transformation and the statement of the correctness theorem.

The syntax and semantics of the languages is 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 is 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. In particular, in Section 2, we let implicit that procedure names should not be duplicated in the list of procedure definitions. This

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

property should be part of the Coq version. Another important property we need to have for programs is that procedure calls are valid, i.e. the name used in the call is indeed a name of a procedure defined in the program. Parallel programs are also labelled. We require that programs given as input to the transformation are correctly labelled. In particular, if they are not, function names generated through the transformation will not be unique, as they use the label of the original statement in the parallel program.

The validity of procedure calls is used to define a relation on procedures. For two procedures p_1 and p_2 of a program $\mathcal{P}$, 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 is necessary to require this property for two reasons. First the simplified way we 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 syntactically sub-terms of one of the arguments of the function. In other cases, a proof of termination should be given.

In particular, the main functions defined to implement the code transformation need such proof of termination. More specifically, one can see from the definition of inlining function, necessary to implement the transformation of atomic blocks, that termination is non trivial. Actually only the fact that $\prec$ is supposed to be well-founded allows to prove it terminates.

As we require some properties in program definitions, we also should prove that the code and memory definitions we obtain from the transformation satisfy the properties of a sequential program. These proofs are conceptually simple, but their implementation in Coq is a bit tedious because they need properties about the functions defined using an explicit proof of termination. This kind of functions is usually heavier to use.

The next step of the mechanization is to define the equivalence between states: having the properties about the uniqueness of procedure names and correct labelling is very important in this regard. The final step will be to prove the correctness. For this we plan to prove semantic preservation for each case of input statement: this will be eased by the fact that the target language is deterministic. This semantic preservation results will then be combined to proof the correctness of the simulation loop where `select` introduces non-determinism.

6 Related Work

Many model checking tools for concurrent programs are based on code sequentialization. In [22], Qadeer and Wu present, for the C language, a transformation from parallel to sequential code that allows the use of existing model checkers for sequential systems. This bounded model checking has been generalized to any context bounds with CSeq [15] and dynamic thread creation [9]. While bounded, such an approach is still efficient to find bugs in concurrent programs [18]. Limiting the comparison to the code transformation, these approaches are a different of ours since in each thread, functions are inlined in the main function, loops are unrolled and that it keeps k copies of the global memory for a bound of k thread context switching.

To avoid creating these copies, allow dynamic memory allocation, and improve perfor-

mances Fisher et al. propose a lazy version of these tools [11, 12] called LazySeq that obtains high performances on known benchmarks. Other research direction choose to bound memory accesses instead of context switching [24].

While efficient to find bugs, these approaches are not suited to prove safety, which is the main reason we aimed at support the Wp plugin of Frama-C. In [19], 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 are now static).

All these approaches consider a sequentially consistent memory model, as we do, other research directions aim at also support weaker behaviors [25, 1].

Why3 is a deductive verification tool [8] 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. Previous versions of Frama-C used Why for deductive verification. Current versions of Frama-C have their own verification condition generator Wp but can still use the translation capabilities of Why3. This is why the work of Fortin and Gava is closely related to ours by the context of the verification framework. They also used program transformation to perform deductive proof of bulk synchronous parallel [26] programs [10]. Why-ML is extended with BSP primitives (including for the assertion language), the resulting language is called BSP-Why-ML. In this work, the original annotated program, written in BSP-Why-ML, is compiled into an equivalent sequential Why-ML program. The deductive proof is then performed using the original Why-ML VCGen, that is 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. 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 communicate with other thread but the result of these communications (message passing) are not effective before the end of the super-step by a synchronization barrier. This constrained form of parallelism as well as the fact that BSP is a distributed memory model, allows a code transformation that is very different from the one we propose.

The way we transform code and specification makes the use of WP after the transformation closely related to Owicki-Gries method [21]. Actually, for each instruction, we have indeed 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 [21], this compatibility is not verified by visiting the proof tree. Owicki-Gries method has been formalized in Isabelle/HOL [20] and one of its variants has been used for verification of operating systems [2, 3]. 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 target simulation program;
- the local environments of the source program should be correctly simulated by the *global heap* of the target 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. For example our method is valid to verify programs under the assumption that procedures are non recursive. Sometimes it may be too limiting. It is therefore interesting to try to lift this limitation. It would require to simulate more precisely the execution context of each thread. This would complexity the correctness proof of the transformation, but we believe the proof would still be manageable, in particular using Coq. However it is unclear whether such an extension would allow the practical analysis of simulating programs and hence of recursive parallel programs. The states of simulating programs would be more complex and the automated provers using in combination to the Frama-C Wp plugin (for deductive verification) may have difficulties to discharge the generated verification conditions.

References

[1] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig. Software verification for weak memory via program transformation. In *ESOP, part of ETAPS*, 2013. doi:10.1007/978-3-642-37036-6_28.

[2] J. Andronick, C. Lewis, and C. Morgan. Controlled Owicki-Gries Concurrency: Reasoning about the Preemptible eChronos Embedded Operating System. In *Proceedings Workshop on Models for Formal Analysis of Real Systems (MARS)*, volume 196 of *EPTCS*, pages 10–24, 2015. doi:10.4204/EPTCS.196.2.

[3] J. Andronick, C. Lewis, D. Matichuk, C. Morgan, and C. Rizkallah. 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*, pages 52–68, 2016. doi:10.1007/978-3-319-43144-4_4.

[4] P. Baudin, J. C. Filliâtre, P. Cuoq, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C Specification Language*, 2015. `http://frama-c.com/download.html`.

[5] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004. doi:10.1007/978-3-662-07964-5.

[6] A. Blanchard, N. Kosmatov, M. Lemerre, and F. Loulergue. 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, pages 15–30, Oslo, Norway, June 2015. Springer. doi:10.1007/978-3-319-19458-5_2.

[7] A. Blanchard, N. Kosmatov, M. Lemerre, and F. Loulergue. 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)*, pages 67–72, Raleigh, NC, USA, 2016. IEEE. doi:10.1109/SCAM.2016.18.

[8] J.-C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus Platform for Deductive Program Verification. In W. Damm and H. Hermanns, editors, *19th International Conference on Computer Aided Verification*, LNCS. Springer, 2007.

[9] B. Fischer, O. Inverso, and G. Parlato. 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*, pages 710–713, 2013. doi:10.1109/ASE.2013.6693139.

[10] J. Fortin. *BSP-Why: a Tool for Deductive Verification of BSP Programs*. PhD thesis, Université Paris-Est Créteil, LACL, october 2013. URL `http://hal.archives-ouvertes.fr/tel-00974977/`.

[11] O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato. 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*, pages 585–602, 2014. doi:10.1007/978-3-319-08867-9_39.

[12] O. Inverso, T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato. Lazy-cseq: A context-bounded model checking tool for multi-threaded c-programs. In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 807–812, 2015. doi:10.1109/ASE.2015.108.

[13] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C: A software analysis perspective. *Formal Asp. Comput.*, 27(3):573–609, 2015. doi:10.1007/s00165-014-0326-7.

[14] N. Kosmatov, V. Prevosto, and J. Signoles. A lesson on proof of programs with Frama-C. Invited tutorial paper. In *TAP*, volume 7942 of *LNCS*, pages 168–177. Springer, 2013. doi:10.1007/978-3-642-38916-0_10.

[15] A. Lal and T. W. Reps. 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*, pages 37–51, 2008. doi:10.1007/978-3-540-70545-1_7.

[16] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Program. *IEEE Trans. Comput.*, 28(9):690–691, 1979. ISSN 0018-9340. doi:10.1109/TC.1979.1675439.

[17] X. Leroy. A formally verified compiler back-end. *Journal of Automated Reasoning*, 43 (4):363–446, 2009. doi:10.1007/s10817-009-9155-4.

[18] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2008, Seattle, WA, USA, March 1-5, 2008*, pages 329–339, 2008. doi:10.1145/1346281.1346323.

[19] T. L. Nguyen, B. Fischer, S. La Torre, and G. Parlato. 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*, pages 174–191, 2016. doi:10.1007/978-3-319-46520-3_12.

[20] T. Nipkow and L. Prensa Nieto. Owicki/Gries in Isabelle/HOL. In J.-P. Finance, editor, *Fundamental Approaches to Software Engineering, Second International Conference (FASE'99)*, volume LNCS 1577, pages 188–203. Springer, 1999.

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

[22] S. Qadeer and D. Wu. 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*, pages 14–24, 2004. doi:10.1145/996841.996845.

[23] The Coq Development Team. The Coq Proof Assistant. `http://coq.inria.fr`.

[24] E. Tomasco, O. Inverso, B. Fischer, S. La Torre, and G. Parlato. 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*, pages 551–565, 2015. doi:10.1007/978-3-662-46681-0_52.

[25] E. Tomasco, T. N. Lam, O. Inverso, B. Fischer, S. L. Torre, and G. Parlato. Lazy sequentialization for tso and pso via shared memory abstractions. In *Formal Methods in Computer-Aided Design (FMCAD)*, 2016.

[26] L. G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103, 1990. doi:10.1145/79173.79181.