CONC2SEQ: A FRAMA-C Plugin for Verification of Parallel Compositions of C Programs

Allan Blanchard*, Nikolai Kosmatov*, Matthieu Lemerre* and Frédéric Loulergue†
*CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette France
Email: firstname.lastname@cea.fr
†Univ Orléans, INSA Centre Val de Loire, LIFO EA 4022, Orléans, France
School of Informatics, Computing, and Cyber Systems, Northern Arizona University, Flagstaff, USA
Email: Frederic.Loulergue@nau.edu

***Abstract*—FRAMA-C is an extensible modular framework for analysis of C programs that offers different analyzers in the form of collaborating plugins. Currently, FRAMA-C does not support the proof of functional properties of concurrent code.**

We present CONC2SEQ, a new code transformation based tool realized as a FRAMA-C plugin and dedicated to the verification of concurrent C programs. Assuming the program under verification respects an interleaving semantics, CONC2SEQ transforms the original concurrent C program into a sequential one in which concurrency is simulated by interleavings. User specifications are automatically reintegrated into the new code without manual intervention. The goal of the proposed code transformation technique is to allow the user to reason about a concurrent program through the interleaving semantics using existing FRAMA-C analyzers.

I. INTRODUCTION

The development of a mature software analysis tool for real-life industrial software is a hard and time-consuming task. This task becomes even more complex for a software analysis platform integrating several tools based on different analysis techniques and capable to collaborate and to benefit from one another's results. Extending such a platform to new features or languages is obviously an ambitious, but very attractive research direction.

FRAMA-C [1] is a popular software analysis platform for C code that offers various static and dynamic analyzers as individual plugins. They include abstract interpretation based value analysis, deductive verification (plugin WP), dependency analysis, slicing, test generation, runtime verification, and many others. FRAMA-C also offers a behavioral specification language ACSL [2] to annotate C programs with contracts. Most FRAMA-C analyzers do not currently support concurrent C code.

For the class of *sequentially consistent* programs [3] (that is, concurrent programs whose execution can be seen as an interleaving of steps of the different threads), the analysis of a concurrent program can be replaced by the analysis of a sequential program that simulates the interleavings of the threads. This simulation-based methodology was applied on a case study in [4]. The corresponding sequential program is generally more verbose and difficult to produce by hand. The main purpose of the CONC2SEQ plugin is to automate this approach and to extend FRAMA-C for analyzing concurrent C code using an automated transformation of concurrent programs into sequential ones.

Contribution. The main contribution of this article includes

- a few new features in FRAMA-C/ACSL facilitating the automation of the simulation-based methodology (support for atomic blocks and thread-local variables, axiomatized atomic primitives, ...)
- an automatic code transformation technique translating a given concurrent code and its specification into a sequential one with an accordingly adapted specification,
- its implementation in CONC2SEQ, a new FRAMA-C plugin allowing the analysis of concurrent C code with FRAMA-C,
- a small case study illustrating the proposed verification methodology and transformation of code and specification, and its proof with FRAMA-C/WP.

The article is organized as follows. First, Section II presents an overview of CONC2SEQ, a running example and the features of ACSL and FRAMA-C we propose for specification of concurrent programs. Then, in Section III, we describe the code and specification transformation technique and the design principles of the CONC2SEQ tool. Section IV compares some existing work to our tool. The advantages and drawbacks of the method are discussed in Section V. Finally, Section VI concludes and gives some future work.

II. ILLUSTRATING EXAMPLE

In this work, we target sequentially consistent concurrent programs in which each of the threads concurrently executes one of the functions `f` $\in F$ of a finite set F. Our main goal is to prove that concurrent execution respects some specified global invariant. The purpose of the CONC2SEQ plugin is to transform an original code into a sequential code that simulates the concurrent behavior of the program and can be then treated in FRAMA-C. The user specifies the desired properties of their program by adding annotations using ACSL, the ANSI C Specification Language [2] offered by FRAMA-C. These annotations are written in special comments `/*@ <annotation> */` or `//@<annotation>`. Such annotations will also be transformed by the plugin. Finally, the plugin will generate a simulating code, that can be processed

```
1  int d, acc;
2  //@ghost int rd __attribute__((thread_local));
3  //@ghost int wr __attribute__((thread_local));
4
5  /*@ logic ℤ sum(ℤ a, ℤ b) = a+b ; */
6
7  /*@ predicate inv =
8        (acc >= -1) ∧ 0 <= rd <= 1 ∧ 0 <= wr <= 1
9      ∧ (acc == -1 ⟺ (1   == th_redux(sum, wr, 0))
10                    ∧ (0   == th_redux(sum, rd, 0)))
11     ∧ (acc >=  0 ⟺ (acc == th_redux(sum, rd, 0))
12                    ∧ (0   == th_redux(sum, wr, 0)));
13     global invariant sw_mr: inv;                    */
14
15 int write(int value){
16   int r, exp = 0;          // previous value must be 0
17   ATOMIC( r = compare_exchange(int, &acc, &exp, -1);
18           /*@ ghost wr = (r == 1) ;*/ );
19   if(! r )                 // ensure write access was granted
20     return 0;
21
22   d = value;                                  // writing d
23
24   ATOMIC( fetch_and_add(int, &acc, 1); // increments acc
25           /*@ ghost wr = 0;*/ );
26   return 1;
27 }
28
29 /*@ requires \valid(l) ∧ \separated(l, &acc, &d); */
30 int read(int* l){
31   int r, a = acc;
32   if(a >= 0){    // previous value must be non-negative
33     ATOMIC( r = compare_exchange(int, &acc, &a, a+1);
34              /*@ ghost rd = (r == 1) ;*/ );
35   }
36   else
37     return 0;
38   if(! r )             // ensure read access was granted
39     return 0;
40
41   *l = d;                                      // reading d
42
43   ATOMIC( fetch_and_sub(int, &acc, 1); // decrements acc
44           /*@ ghost rd = 0;*/ );
45   return 1;
46 }
```

Fig. 1. Single writer / multiple readers policy example

by FRAMA-C plugins. In this work we focus on its proof with the deductive verification tool FRAMA-C/WP. The generated code is also printed into a user-specified file. That allows the user to insert additional annotations into the simulating code that can be necessary for its proof.

Figure 1 presents a running example implementing a single-writer, multiple-readers policy to access a particular data resource. Several threads concurrently execute functions `read` or `write` to access the value of a shared variable `d`. The parameter `l` is assumed to refer to a *valid* memory location (i.e. that can be safely read/written), *separated* from (i.e. non-aliased by) `acc` and `d` (line 29). The access variable `acc` associated to `d` is used to store the current access status: its value is 0 for no access, -1 for write access (by a unique thread) or some $k > 0$ for read access (by one or several threads). In case of several reading threads, the value of `acc` contains the number of threads currently having read access to `d`. If a thread that intends to write into `d` succeeds in changing the value `acc` from 0 to -1, it gains exclusive write access. If a thread that intends to read `d` succeeds in incrementing the value of `acc` and this value was not negative before (i.e. nobody is writing), it gains read access. A thread gives back the granted access by incrementing (resp. decrementing) the `acc` variable to lose write (resp. read) access. We assume that `acc` does not overflow, i.e. the number of simultaneous threads is bounded. The property that we want to ensure is mutual exclusion between read and write accesses.

Atomic primitives. We have specified some standard functions for atomic memory operations in ACSL so that deductive verification can prove examples using such functions. In our example, the modifications of `acc` are performed by functions `compare_exchange`, `fetch_and_add` and `fetch_and_sub`. The latter two ones are used to increment or decrement the value of `acc` (cf. lines 24, 43).

An instruction `r=compare_exchange(int,&acc,&a,x)` (cf. lines 17, 33) can be seen as an atomic version for: `if(acc==a){ acc=x; r=1; } else { a=acc; r=0; }` that checks if `acc` contains the expected value `a` and replaces it by the new value `x` in this case. Otherwise, it does not change it and returns the actual value of `acc` in `a`.

Ghost code. To specify the global invariant, it can often be convenient to use *ghost* (or *phantom*) code, introduced in ACSL annotations using the keyword **`ghost`**. As for all other ACSL annotations, ghost code is visible only by the verification tool and is ignored during program compilation and execution. Ghost code is a way to more precisely or explicitly define the program current state in order to help the verification tool.

In our example, we need to specify the number of readers and writers in a way suitable for formal proof. We choose to use two ghost variables, `wr` and `rd` (cf. lines 2–3), that will explicitly record whether the current thread is a writer (`wr`$=1$, `rd`$=0$), a reader (`wr`$=0$, `rd`$=1$), or none of those (`wr`$=0$, `rd`$=0$). These variables are *thread-local*, that is, they are global but visible only in the current thread. We add a support for thread-local declarations in CONC2SEQ.

When a thread gets an access as a writer (resp. reader), it puts 1 (cf. lines 18, 34) into its thread-local variable `wr` (resp. `rd`). When it returns its access, the thread puts 0 into the corresponding variable (cf. lines 25, 44). To avoid an incoherent representation of thread statuses in concurrent execution, an important point is that the ghost variables have to be updated in the *same atomic step* used to update the access variable. We extend ACSL by the possibility to specify a block of instructions as atomic using a special annotated block, or simply by a macro `ATOMIC` (that will build such a block). Here we illustrate the use of the macro (e.g. lines 17–18). The only restriction using this kind of annotation is that it is forbidden to jump (**`goto`**) outside of the block from inside the block, as well as to jump into the block from outside. Otherwise, it would break the semantics of an atomic block that should be executed as one indivisible step.

Reduction over all threads. The exclusion property can be stated by two exclusive cases: either exactly one of `wr` over all threads is equal to 1 and all `rd` are 0; or all `wr` are equal to 0 and any number of the `rd` can be equal to 1. Thus, to state the global invariant we need to count how many writers and readers we have at each moment of execution. In order

to facilitate the specification of such concurrent properties, we extend ACSL by a new built-in logic function that lets the user specify properties for the set of values of a certain local or thread-local variable over all threads. This function is called `th_redux`, cf. lines 9–12. Like a "fold" in functional programming languages, it takes 3 parameters: the function to apply on each new element and the previously computed result, the value on which we want to apply it over all threads and the initial value from which we start the computation. Here we use it to express, on the logic side, the number of threads that have read or write accesses. For instance, the expression `th_redux(sum, wr, 0)` computes the number of writers as the sum of `wr` values over all threads. (Concretely, for two threads with values `wr1` and `wr2`, it will compute `sum(wr1,sum(wr2,0))`.)

Global invariant. The global invariant that we want to maintain during concurrent execution is specified with the ACSL clause **`global invariant`** using the predicate `inv` (lines 7–13). It defines admissible values for `acc`, `rd` and `wr`, and states that `acc` is -1 if and only if the number of writers is exactly 1 and the number of readers is 0; and that `acc`≥ 0 if and only if `acc` is exactly the number of readers and the number of writers is 0. This statement implies the mutual exclusion of read and write accesses.

III. Design Principles

This section presents the transformation technique we use to construct a simulating sequential code for a given concurrent program, and describes the design of the Conc2Seq tool that automatically performs this transformation.

A. Modeling of execution context

The considered execution context includes global variables, thread-local global variables, local variables and the program counter. In the simulating code, global variables are kept as they are. Each function parameter, each local or thread-local variable is transformed into an array that associates to a thread identifier the value of the corresponding variable in this thread. Another array, called `pct`, models, for any thread, the value of its program counter, i.e. which atomic step is executed by the thread. In our example, the variables are simulated as illustrated in Figure 2 with the program counter (line 1), the thread-local variables (line 2), the local variables of `read` (line 3) and the local variables of `write` (line 4). For instance, `gl_rd[2]` and `pct[2]` represent the value of ghost variable `rd` and the program counter of the thread of index 2, while `read_a[2]` is the value of local variable `a` in function `read` if this thread executes this function (and undefined otherwise).

As we do not allow dynamic thread creation, we assume that the number of threads is bounded, and model it as a logic value `MAX_THREADS` which is only assumed to be greater than 0. Consequently, the arrays modeling the execution context are not static fixed-size arrays and should be defined as pointers. We automatically generate necessary ACSL annotations to assume axiomatically that each of them points to valid elements for any valid thread index (cf. lines 15–16), i.e. from index 0 to index `MAX_THREADS-1`. We also axiomatically state that these memory blocks are not aliased with each other, nor with existing global variables (cf. line 11).

```
1  int *pct;
2  int *gl_rd, *gl_wr;
3  int *read_r, *read_a, **read_loc;
4  int *write_r, *write_exp, *write_value;
5
6  /*@ axiomatic Validity_of_simulating_vars {
7    predicate simulation{L} reads <all simulation pointers> ;
8
9    axiom all_simulations_separated{L}:
10   simulation ==>
11     \separated( <all simulating memory blocks/globals> );
12
13   axiom pct_is_valid{L}:
14     simulation ==>
15       (∀ ℤ j; valid_th(j) ==> \valid(\at(pct,L)+j));
16   //same kind of axioms for each simulating pointer
17   //...
18 } */
```

Fig. 2. Simulation of execution context

Let us emphasize one technical issue that can also be addressed by automatic generation of annotated code. If such an axiom naively states validity or separation without giving any information on memory locations referred to by these pointers (i.e. stating just line 11 or 15), the axiom will lead to a proof of false because it is obviously not true for any memory location (for example, separation is broken if `pct==&d`). Indeed, the truth value of these properties relies on the fact that the corresponding pointers are appropriately defined. To constrain the definition of simulating pointers (that cannot be statically bound to static fixed-sized arrays), we add an abstract (undefined) ACSL predicate `simulation` that depends on the values of the simulating pointers (line 7). Then we state that the separation (lines 9–11) and validity (lines 13–15) must be ensured whenever `simulation` is verified. This predicate will be part of the global invariant of the simulating code that will constrain simulating pointers and ensure that the axioms remain meaningful during the verification.

B. Normalized AST

Thanks to the CIL library [5], Frama-C creates and normalizes the input program's Abstract Syntax Tree (AST) and computes the control flow graph (CFG). In particular, side-effects are moved outside of expressions, conditional statements with compound conditions are unfolded into multiple conditionals (with one non-compound condition each), while loops are replaced by equivalent **`while`**`(1)` loops (with additional conditional and break statements to leave the loop body). So, as most Frama-C plugins, Conc2Seq can just rely on the normalized AST. The AST assigns unique identifiers for statements and blocks that can be used for the program counter. For the sake of clarity, we use line numbers as identifiers in this paper.

In a first analysis pass, Conc2Seq can transform the AST by creating a load statement to a new temporary local variable for every access to the global memory or simply through a pointer. As a result, every statement contains at most one global memory access, and compound expressions can only

```
1  /*@ requires valid_th(th) ∧ *(pct+th) == 22 ;
2      requires simulation ∧ inv ;
3
4      ensures *(pct+th) == 24;
5      ensures  simulation ∧ inv ; */
6  void write_Instr_22(unsigned th){
7    d = *(write_value + th);
8    *(pct + th) = 24;
9    return;
10 }
11
12 /*@ requires valid_th(th) ∧ *(pct+th) == 32 ;
13     requires simulation ∧ inv ;
14
15     ensures *(pct+th) == 33 ∨ *(pct+th == 37);
16     ensures  simulation ∧ inv ; */
17 void read_If_32(unsigned th){
18   if (*(read_a + th) >= 0)  *(pct + th) = 33;
19   else                      *(pct + th) = 37;
20   return;
21 }
```

Fig. 3. Simulating functions for atomic steps at lines 22 and 32 of Fig. 1

involve local variables so that they can be considered atomic from the shared memory point of view.

C. Atomic steps, function entries and interleavings

In the normalized AST, every individual statement is supposed to be an atomic step, as well as every block specified as "atomic" by the user. Each atomic step is modeled by a simulating function that takes as a parameter the number of the thread that executes this step. The basic idea in the modeling of an atomic step is to perform exactly the same operation except that every access to a local or thread-local variable is replaced by an access to the simulating array element corresponding to this variable and this thread. Once the step has been performed, the program counter of the thread is updated to the identifier of the next step(s) to be performed (for a conditional statement, we have two choices depending on the condition evaluation). Figure 3 illustrates the result of transformation for statements at lines 22 and 32 of Figure 1.

We currently support the following statement types: assignments, atomic function calls, return, if/else, switch statements, atomic blocks and loops. Break, continue, non-atomic block entrance and goto statements are "inlined", in the sense that we directly set the program counter to the statement they jump to, recursively if it is also an unconditional jump. So for such statements we do not generate a dedicated simulating function since it would only modify the program counter without any impact on the global or local program state.

Unspecified C sequences are detected by FRAMA-C and signaled as errors by CONC2SEQ. We do not currently support assembly code, nor calls to non-atomic functions in order to keep our modeling of the context simple. We plan to support calls to recursion-free non-atomic functions in the future.

For every function $\texttt{f} \in F$ that can be executed by a thread, a separate step simulates the beginning of execution of `f`. We generate a function declaration `init_f` with a contract that models this step for `f`. In order to ensure that the precondition of `f` is respected in the simulating code, we state it as a postcondition of function `init_f`. Somewhat, it models the function context initialization. By doing this, the simulated

```
1  /*@ requires valid_th(th) ∧ *(pct+th) == -30;
2      requires simulation ∧ inv ;
3
4      ensures *(pct+th) == 31;
5      ensures simulation ∧ inv ;
6
7      ensures \valid(*(read_l+th))
8           ∧ \separated(*(read_l+th), &acc, &d);*/
9  void init_read(unsigned int th);
```

Fig. 4. Modeling a call to `read` in the simulating code

```
1  /*@ requires simulation ∧ inv ; */
2  void interleave(void)
3  {
4    unsigned int th;
5    th = some_thread();
6    /*@ loop invariant simulation ∧ inv ; */
7    while (1) {
8      th = some_thread();
9      switch (*(pct + th)) {
10       case 0   : choose_call(th);    break;
11       case -15 : init_write(th);     break;
12       case -30 : init_read(th);      break;
13       case 32  : read_If_32(th);     break;
14       case 22  : write_Instr_22(th); break;
15       //... similar cases for other atomic steps
16     }
17   }
18   return;
19 }
```

Fig. 5. Simulating concurrent execution by interleavings

entry to `f` simply consists in positioning formal parameters of `f` (in the considered thread) to values that respect the precondition of `f`. In this work we do not need to define the code of `init_f` because in modular deductive verification, a specified C prototype is sufficient to verify its caller. Function identifiers are negative (defined as negated line numbers in this paper). For example, Figure 4 illustrates the step simulating the entry of `read`, where the postcondition at lines 7–8 is exactly the precondition of `read` (Figure 1, line 29).

Once all simulating functions for atomic steps and function entries are created, we model interleavings by an infinite loop that at each step, selects a random valid thread number and calls the simulating function corresponding to the next step it has to perform. We illustrate it in Figure 5, where we only mention the simulating functions presented in this section.

D. Specifications

Automatic generation of specifications for the simulating code has two important concerns: translation of user specifications of the original code and adding new specifications necessary to define the simulation itself.

User specifications can be of three types: function contracts (including pre- and postconditions), global invariants, and assertions associated to particular program points. The support of function preconditions is a key feature described in Sec. III-C. As we mainly aim at verifying global invariants, the support of postconditions is not mandatory (we plan to add them as a postcondition to the simulated return statement). The support of assertions can be also left as future work.

To ensure preservation of global invariants, we collect and insert them into the contract of every simulating function both as a pre- and a postcondition, as well as into a loop invariant of the interleaving loop (cf. Figures 3, 4, and 5). If

an original global invariant invokes a thread-local variable, in the generated specification of the simulating code we replace it with an access to the simulating global array (usually such properties state a relation between all of them, cf. Fig. 1, lines 9–12).

To ensure a correct control flow in the simulation, we specify for every simulating function a precondition (resp. postcondition) stating the current (resp. next) value(s) of the program counter. We also add an additional predicate to the global invariant `inv` to ensure that the program counter is indeed the identifier of a valid instruction.

Another helpful feature for automation of the simulation-based verification methodology is the support we propose for the builtin function call `thread_redux(h,v,b)` (cf. Sec. II). We generate both a first-order axiomatic specification that inductively defines the result of this call as well as some classic lemmas that can be used for the proof of more complex properties invoking such calls in FRAMA-C/WP. Technically, since FRAMA-C/WP does not currently support higher-order specifications, we generate a new version for each new function `h` and each new type of `v` (where the function `h` is inlined and not any more a parameter in the definition). Then, for any call, we replace the builtin call by a generated function call where the original variable is replaced by the simulating one.

Considering standard atomic routines, they are generally implemented as macros (rather than functions) and require to indicate the type of the considered variables as parameters. Therefore, in order to specify such primitives for modular verification, we define new macros that define, for each used type, a new function prototype and the desired specification for this primitive. The user should explicitly indicate the manipulated type in the call of an atomic routine as it is shown in Figure 1, lines 17, 24, 33 and 43.

IV. RELATED WORK

We designed the CONC2SEQ tool to automate the method used in [4] to prove concurrent functions of a microkernel. This method appears to be useful for proving concurrent properties in isolation from the whole system, but quite tedious and error-prone to apply manually. The automation removes the tricky part, easing the process of verification. Most of the simulating code written by hand can now be generated.

The way we transform code and specification makes the use of WP after the transformation closely related to Owicki-Gries method [6]. 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 [6], this compatibility is not verified by visiting the proof tree. Owicki-Gries method has been formalized in Isabelle/HOL [7] and one of its variants has been used for verification of operating systems [8]. So, even if it can generate a lot of verification conditions, it is still usable in practice for real-life code.

Rely/Guarantee [9] reasoning allows to specify and prove concurrent programs more modularly than Owicki-Gries method, by providing a way to specify how threads are allowed to modify the global state of the program. "Relying" on this knowledge, desired properties are locally proven, and "Guarantees" are given about modifications performed on the global state. It is implemented for example in Isabelle/HOL [10]. It could be interesting to study whether this methodology can allow us to limit the amount of generated specifications.

VCC analyzer [11] is dedicated to verification of concurrent C. It relies on the idea that every data-structure has an invariant that has to be maintained by all actions on it ("stable invariant") and by stuttering steps ("reflexive invariant"). The ability to modify objects is handled by a notion of ownership. Boogie [12] is used to perform the verification by weakest precondition calculus. CONC2SEQ has the additional goal to allow not only the use of WP but also of other FRAMA-C plugins such as abstract interpretation or code instrumentation. We think code transformation is a fast way to provide it.

A program transformation based tool in [13] allows concurrency-aware analyzers to reason under weak behaviors. It transforms an original concurrent code into a new one which is still concurrent but where weak behaviors are explicit. We want to allow analysis of concurrent programs using analyzers that are not concurrency-aware. As we only reason about programs under interleaving semantics, it would be interesting to see whether we can extend our support for concurrency to programs under weak behaviors.

V. DISCUSSION

One important prerequisite of our work is to analyze a program that is sequentially consistent. In fact, the C norm indicates that a racy program contains *undefined behavior* [14, Sec. 5.1.2.4]. Hence, to be correct according to the norm, a C program must be data-race free, and such programs are sequentially consistent [15]. Data-race freedom could be ensured by a different analysis step. One can for example use the FRAMA-C plugin MTHREAD [1, Sec. 9] to check that the program is data-race free, and then use CONC2SEQ to prove its functional properties.

Currently, when we generate simulating function specifications, we do not keep relations between local variables. On a simple program like `int x = 1; int a = x;`, we generate two simulating functions. But in the simulation of the second instruction, we do not generate any precondition stating that `x` is equal to`1`. So we are unable to prove locally that `a` becomes `1`. Currently, the user has to manually add such specifications herself in the generated code.

This kind of relations, expressing properties about local variables satisfied for some values of the program counter of the simulating program, could be generated using a simple forward analysis on the simulated function. For example, symbolic execution of the function can collect known relations between variables at each statement. Then these relations could be translated into ACSL annotations according to variable simulation, and placed as a postcondition of the preceding statement and a precondition of the next one. In the same way, additional user assertions (that can remain necessary,

for instance, for loop invariants) could also be transformed. Special care should be taken to drop properties that involve access to the global memory as it can be modified by other threads from one instruction to another.

To model the execution context, we use pointers to axiomatically valid memory blocks. Compared to C static arrays, it does not give memory separation "for free", and we have to state that the validity of the memory blocks is dependent of the simulating pointers. It would be more practical to use static arrays of undefined size and just state axioms about range validity (separation and pointer validity come "for free"). WP, which is the first plugin we target to support, does not currently handle this type of arrays. For this particular target, adding the support of undefined size arrays would be particularly interesting as it would allow a better translation to SMT solvers. We could also add an option to generate classic C static arrays with a user-specified size.

The extraction of the simulation into a file is practical to finish the proof. The names are already generated, and the functions are organized in a readable way making it easier to write additional specifications for the generated code. For example, some second order properties require to add specifications at the level of the generated code because the reduction over all threads built-in is not suited to every possible property. Moreover, when proofs are not discharged automatically, additional assertions may guide the proof search.

For example, to complete the proof of the code in Figure 1 with WP, we need to add some assertions. Most of the proof objectives are automatically proved once relations between locals are correctly provided along with a few assertions in simulating functions. On a similar code (where we have split the invariants), we generate 718 proof obligations, comprising 441 for the `interleave` function whose proof is trivial (direct use of function contracts). 704 obligations are automatically proved with Frama-C Aluminium using Alt-Ergo 1.01 and Z3 4.4.2. It takes 260s on a QuadCore Intel Core i7-4800QM @2.7GHz. The 14 remaining proofs (about the sum of writers/readers) are a bit harder as they require to avoid the complete induction reasoning. It is done with help of assertions needed by the lemmas generated for the built-ins.

VI. Conclusion and Future Work

We have implemented a new FRAMA-C plugin, CONC2SEQ, for analysis of concurrent C code with existing plugins. Currently we focus on WP to take advantage of SMT solvers for automatic deductive proof of functional properties. We have also provided some new ACSL constructs to specify properties involving the state of multiple threads.

The (mostly) syntactic transformation into equivalent C code simulating the concurrent behaviors of the program produces comprehensive generated code, that can be conveniently completed by additional assertions in order to finish its verification.

In future work, we plan to perform a formal proof that the transformation is correct using the Coq proof assistant. This proof relies on a simplified language that captures the key notions that impact the semantics: concurrent execution and memory accesses as well as basic control structures. The transformation implemented in Coq follows that of CONC2SEQ.

From the point of view of provided features, we want to add an analysis step that collects relations between local variables and user-provided assertions at each program point in order to translate them into pre- and postconditions of the simulating functions. It would also be interesting to enrich ACSL with new built-ins to specify properties of concurrent behaviors.

Currently, the way we perform the transformation is focused on the use of the deductive verification plugin WP on the generated code. Some constructs we use are not handled by other important plugins of FRAMA-C, especially the value analysis or runtime verification plugins. It would be interesting to bridge the gaps to them. It would also be useful to add the support of static array declarations with a locally undefined size to the WP plugin.

Acknowledgement. This work has received funding for the S3P project from French DGE and BPIFrance and for the Ph.D. grant of the first author from French Ministry of Defence. Thanks to the anonymous referees for their helpful comments.

References

[1] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski, "Frama-c: A software analysis perspective," *Formal Asp. Comput.*, vol. 27, no. 3, pp. 573–609, 2015.

[2] 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.

[3] L. Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Program," *IEEE Trans. Comput.*, 1979.

[4] 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 *FMICS*, 2015.

[5] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer, "CIL: intermediate language and tools for analysis and transformation of C programs," in *CC*, 2002.

[6] S. Owicki and D. Gries, "Verifying properties of parallel programs: an axiomatic approach," *Communications of the ACM*, 1976.

[7] T. Nipkow and L. Prensa Nieto, "Owicki/gries in Isabelle/HOL," in *FASE*, 1999.

[8] J. Andronick, C. Lewis, and C. Morgan, "Controlled Owicki-Gries Concurrency: Reasoning about the Preemptible eChronos Embedded Operating System," in *MARS*, 2015.

[9] C. B. Jones, "Tentative steps toward a development method for interfering programs," *ACM Trans Program Lang Syst*, vol. 5, 1983.

[10] L. Prensa Nieto, "The Rely-Guarantee Method in Isabelle/HOL," in *ESOP*, 2003.

[11] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "VCC: A practical system for verifying concurrent C," in *TPHOLs*, 2009.

[12] M. Barnett, B.-Y. E. Chang, R. DeLine, B. Jacobs, and K. R. M. Leino, "Boogie: A Modular Reusable Verifier for Object-Oriented Programs," in *FMCO*, 2005.

[13] J. Alglave, D. Kroening, V. Nimal, and M. Tautschnig, "Software verification for weak memory via program transformation," in *ESOP, part of ETAPS*, 2013.

[14] International Organization for Standardization, *ISO/IEC 9899:2011: Programming languages – C*. ISO Working Group 14, 2011.

[15] V. A. Saraswat, R. Jagadeesan, M. M. Michael, and C. von Praun, "A theory of memory models," in *PPoPP*, 2007.