Software Analyzers

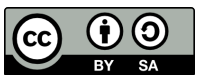
Aoraï Plugin Tutorial

(A.k.a. LTL to ACSL)

For Frama-C 34.0 beta (Selenium)

Nicolas Stouls and Virgile Prevosto
Nicolas.Stouls@insa-lyon.fr,virgile.prevosto@cea.fr





Work licensed under Creative Commons BY-SA licence
https://creativecommons.org/licenses/by-sa/4.0/

© 2009-2026 CEA-List

Foreword

Aoraï is a Frama-C plugin that provides a method to automatically annotate a C program according to an automaton F such that, if the annotations are verified, we ensure that the program respects F. A classical method to validate annotations then is to use Eva or WP.

This document requires basic knowledge about the Frama-C platform itself (See https://frama-c.com for more information), in particular the notions of *plug-ins* and *project*.

Notes:

– to the question "Why this name: *Aoraï*?" my answer is: why not? Aoraï is the name of the tallest reachable mount in the Tahiti island and its reachability is not always obvious.

Official website:

https://www.frama-c.com/fc-plugins/aorai.html

Contents

1 **Introduction** 5

1.1 Installation . . . 5

1.2 Interest of Aoraï . . . 5

1.3 Documentation's description . . . 5

2 **Quick overview** 6

2.1 First use . . . 6

2.1.1 Launching the test . . . 6

2.1.2 Automata and verification . . . 7

2.2 Help Command . . . 7

2.3 Known Restrictions . . . 8

3 **Aoraï's Language** 11

3.1 YA . . . 11

3.1.1 YA file . . . 11

3.1.2 Basic YA guards . . . 12

3.1.3 YA extensions . . . 13

4 **Advanced Features** 16

4.1 Generated Annotated Program . . . 16

4.1.1 Auxiliary Variables . . . 16

4.1.2 Deterministic lemmas . . . 16

4.1.3 Update functions . . . 17

4.1.4 Functions behaviors . . . 19

4.1.5 Loop Invariants . . . 19

4.2 Analyzing Annotated Files with Eva or WP . . . 20

5 **Going Further** 22

5.1 Theoretical Base of the Approach . . . 22

5.1.1 Safety . . . 22

5.1.2 Liveness . . . 23

5.2 Adding from the Theory . . . 23
5.2.1 Automata Modeling . . . 23
5.2.2 Memorization of last Transitions . . . 23
5.2.3 Use of Specifications instead of Invariant . . . 23
5.3 Abstract Interpretation . . . 24
5.3.1 Generation of Abstract Specifications . . . 24
5.3.2 Static Simplification . . . 24
5.4 Plugin Architecture . . . 24
5.5 Recent updates . . . 25
5.5.1 Frama-C Iron . . . 25
5.5.2 Frama-C Vanadium . . . 25
5.5.3 Frama-C Titanium . . . 25
5.5.4 Frama-C Aluminium . . . 25
5.5.5 Frama-C Nitrogen . . . 25
5.5.6 Frama-C Boron . . . 25
5.5.7 Frama-C Beryllium . . . 26

6 **Conclusion** 27

Introduction 1

1.1 Installation

Aoraï is part of the main Frama-C distribution and is installed by default with Frama-C. Installation instructions for Frama-C are available at https://frama-c.com/html/get-frama-c.html.

1.2 Interest of Aoraï

As explained before, Aoraï's goal is to prove that the C program works like a given automaton. The approach used by Aoraï has two advantages:

- the high level of abstraction helps to write simple automata and avoid the necessity to compute all possibilities of a function[1]
- thanks to the collaboration between human and plugin principle, you can easily check complex C programs (see section 4.2)

1.3 Documentation's description

This document is divided into four parts:

- First part is a quick overview of Aoraï. It will enable you to verify basic properties and explain the general principle of the software.
- The second part defines the Aoraï input language with which it is possible to describe a given property.
- The third part explains how to prove a program annotated with Aoraï using the WP plug-in.
- Finally, the last part details Aoraï's underlying theory, and its internal architecture in order to help people who would like to contribute to the plug-in itself.

[1] for more information, see chapter 5

Quick overview 2

In this chapter we will see how to use Frama-C and the couple WP-Aoraï to prove that a C program has the same behavior as an automaton.

2.1 First use

The goal is to launch the examples[1] and read results.

2.1.1 Launching the test

First, we will forget about the specification of the automaton, which will be described in the second part. In fact, we consider that we have already written the file which describes the automaton.

WP verification[2] can only be done on C code augmented with ACSL annotations. Thus, Aoraï creates a new C file where the automaton is encoded into ACSL annotations. Section 4.1 will give more information about the annotations generated by Aoraï

If you look at the example's archive, you will find two files:

- `example.ya` which gives a description of the automaton's specifications.
- `example.c` is the implementation which will be checked.

With these two files (automaton's description and C file), we can create an annotated file. This is done by the following command:

```
$ frama-c example.c -aorai-automata example.ya \
        -then-last -ocode example_annot.c -print
```

This generates a new C file `example_annot.c`. In order to decide if the original program is correct with respect to the automaton, it is sufficient to establish that the generated C code and its associated ACSL annotations are valid. For instance, the following command uses the WP plugin over the generated file:

```
$ frama-c example_annot.c -wp -wp-rte
```

Of course, any option of WP itself can be used, notably `-wp-rte` to check for the absence of runtime error. Finally, it is possible to instruct Frama-C to do a sequence of analyzes over various projects, *via* the `-then-on` options and `-then-last` options. Thus, we do not need to use an intermediate file and to run Frama-C twice. Instead, we just instruct WP to operate on the `aorai` project that contains the code annotated by Aoraï:

[1] From http://frama-c.com/aorai.html

[2] For more information about WP and code verification, please refer to http://frama-c.com/wp.html

```
$ frama-c example.c -aorai-automata example.ya \
        -then-last -wp -wp-rte
```

2.1.2 Automata and verification

The main interest of Aoraï is to prove that the program can be described by an automaton. Please keep in mind that solutions to write automata in Aoraï are listed in the next chapter.

The automaton of our running example is described by figure 2.1.

From the description contained in `.ya` file, a specification — in terms of automata states and transitions — is computed for each operation. For instance, the following specification corresponds to the previous automaton:

$$\text{opa} \begin{cases} \text{Pre}: & state = \{2\} \ \wedge \ trans = \{1\} \\ \text{Post}: & \backslash old(state) = \{2\} \Rightarrow state = \{3\} \ \wedge \ trans = \{2\} \end{cases}$$

$$\text{opb} \begin{cases} \text{Pre}: & state = \{4\} \ \wedge \ trans = \{3\} \\ \text{Post}: & \backslash old(state) = \{4\} \Rightarrow state = \{5\} \ \wedge \ trans = \{4\} \end{cases}$$

$$\text{opc} \begin{cases} \text{Pre}: & state = \emptyset \ \wedge \ trans = \emptyset \\ \text{Post}: & \backslash old(state) = \emptyset \Rightarrow state = \emptyset \ \wedge \ trans = \emptyset \end{cases}$$

$$\text{main} \begin{cases} \text{Pre}: & state = \{1\} \ \wedge \ trans = \{0\} \\ \text{Post}: & \backslash old(state) = \{1\} \Rightarrow state = \{6\} \ \wedge \ trans = \{5\} \end{cases}$$

Finally, the C-code which will be checked is given in figure 2.2.

Actually, the mapping between state and code is made thanks to the transitions properties like *CALL(opa)*. Note that the pre- and post-conditions of the C functions are defined by the set of states authorized just before (resp. after) the call.

Aoraï generates a new C program, including the automaton axiomatization, some coherence invariants, and annotations on operations, such that if this annotated program can be validated with the WP plugin, then we ensure that it respects the given properties.

Sometimes, the automaton has not enough information to check the validity of the C-program, and the problem is only related to the implementation which is used. In this case you can add some properties in the automaton *or* in the generated files. For more information about that, please read section 4.2.

2.2 Help Command

The `frama-c -aorai-help` command returns the list of options for the Aoraï plug-in. Here are the most common ones:

-aorai-automata <f> considers the property described by the ya automata (in Ya language) from file `<f>`

-aorai-verbose <n> gives some information during computation, such as used/produced files and heuristics applied

-aorai-show-op-spec displays, at the end of the process, the computed specification of each operation, in terms of states and transitions.

-aorai-dot generates a dot file of the automata. Dot is a graph format used by the GraphViz tools[3].

[3] http://www.graphviz.org

2.3 Known Restrictions

The current version of Aoraï is under development. Hence, there are some restrictions.

- Only safety properties can be checked. The liveness part is not truly considered. Namely, the acceptance condition is a post-condition of the main function, so that we need a terminating program to actually validate it: for a non-terminating program, the point where it is supposed to be checked is unreachable.
- Function pointers are only supported if each indirect call comes with an `calls` ACSL annotation (see Frama-C user manual for more information)
- In the init state from the automaton, conditions on C-array or C-structure are not statically evaluated (it's an optimization) but are supported.

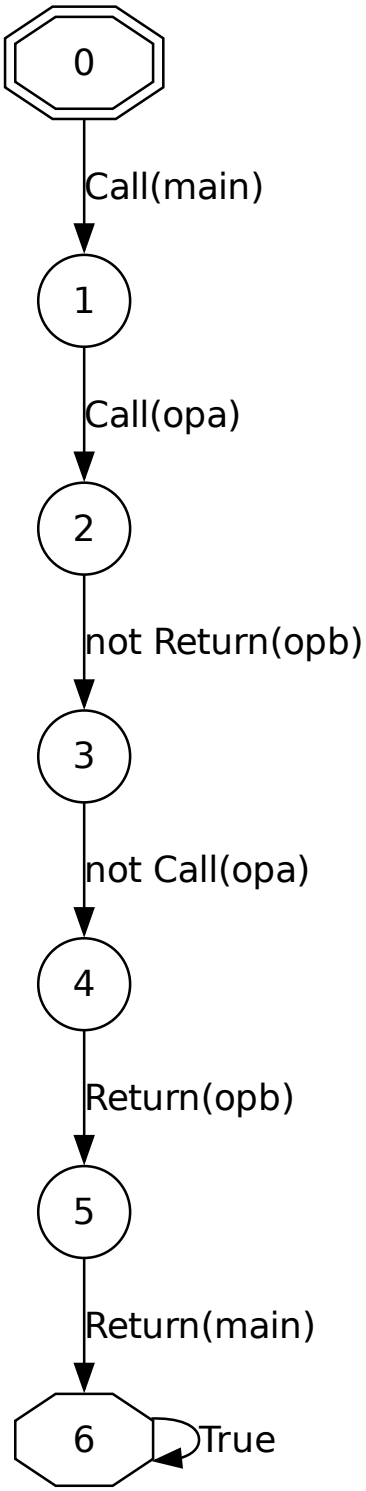


Figure 2.1: Automaton

```
int rr=1;
//@ global invariant inv:0<=rr<=5000;

/*@ requires r<5000;
  @ behavior j :
  @  ensures \result==r+1;
*/

int opa(int r) {return r+1;}

/*@ requires rr>=1 && rr <=5000;
  @behavior f:
  @   ensures rr>=3 && rr<=5000;
*/
void opb () {if(rr<4998) {rr+=2;}}
/*@ behavior d:
  @   ensures rr==600;
*/
void opc () {rr=600;}

/*@ requires rr==1;

*/
int main() {
  if (rr<5000) rr=opa(rr);
  opb();
  goto L6;
  opc();
 L6:
  return 1;


}
```

Figure 2.2: Example of C File

Aoraï's Language 3

Aoraï's verification principle is built from the automaton. Automata can be described in the YA language, which was created for Aoraï.

3.1 YA

3.1.1 YA file

A YA file follows the grammar described in Fig. 3.1. The directives specify the initial and accepting

file	::=	*directive** *state*$^+$	
directive	::=	**%init** : *id*$^+$;	name of initial state(s)
	|	**%accept** : *id*$^+$;	name of accepting state(s)
	|	**%deterministic** ;	deterministic automaton
state	::=	*id* : *transition*	
		(| *transition*)* ;	state name and transitions from state
transition	::=	*guard* -> *id*	guard and end state of the transition
	|	-> *id*	transition that is always taken
	|	**other** -> *id*	default transition. must appear last
guard	::=	{ *condition* }	

Figure 3.1: Structure of a YA file

state(s). There must be at least one initial state (exactly one if the automaton is supposed to be deterministic). All initial and accepting state must appear in the list of states afterwards.

A state is simply described by its name and the list of transitions starting from this state with their guard. The specific **other** guard indicates that this transition is taken if none of the other ones can be taken. If it appears, it must be last in the list of transitions.

```
condition ::= CALL ( id ) | RETURN ( id )
            | COR ( id )
            | true | false | ! condition
            | condition && condition
            | condition || condition
            | ( condition ) | relation

relation  ::= expr relop expr

relop     ::= < | <= | == | != | >= | >

expr      ::= lval | cst | expr + expr | expr - expr
            | expr * expr | expr / expr | expr % expr
            | ( expr )

cst       ::= integer

lval      ::= id ( ) . id | id ( ) . \result | id
            | lval . id | lval -> id
            | lval [ expr ] | * lval
```

Figure 3.2: Basic YA guards

Conditions that can occur in guards are described in the next section.

3.1.2 Basic YA guards

The syntax for basic YA conditions is described in figure 3.2.

Basically, a condition is a logical expression obtained from the following atoms:

- **CALL**, **RETURN** or **COR** event, indicating respectively the call, the return, the call or the return of the corresponding function;
- A relation over the variables of the programs. In addition to global variables, that are directly accessed through their `id`, it is possible to consider the value returned by a function or the value of its formal parameters. This is done through `f().return` and `f().a` respectively. In order to be closer to ACSL's syntax, `f().\result` is accepted as a synonym of `f().return`.

Whenever `f().prm` appears in a relation, the related guard has an implicit **CALL**`(f)` event, while `f().return` and `f().\result` trigger a **RETURN**`(f)` event. Note that this might result in an always-false guard if several such expressions occur in the same guard, as in

```
f().x <= g().y
```

In order for this guard to hold, we should be calling at the same time `f` and `g`, which is not possible. In addition, if such expression occurs in a negative occurrence, that is under a negation, as in

```
! f().x <= 4
```

the related **CALL**(f) event itself is *not* negated. In other words, the guard above is true if and only if we call f with an argument greater than 4. Usage of these expressions might be deprecated in future versions of Aoraï in favor of the less ambiguous constructions presented in the next subsection.

For instance, the automaton used in the chapter 2.1 contains the following transitions:

```
%init: S0;
%accept: S0, S1, S2,S3,S4,S5,S6;
S0 : { CALL(main) }  -> S1;
   ;
S1 : { opa().r<5000 }   -> S2
   ;
S2 : { opa().return<=5000 }  -> S3
   ;
S3 : { !RETURN(opa) } -> S4
   ;
S4 : { RETURN(opb) } -> S5
   ;
S5 : { RETURN(main)} -> S6
   ;
S6 : -> S6
   ;
```

3.1.3 YA extensions

Extended YA guards

In order to describe more easily whole sequences of calls, some extensions to the basic conditions above are available. They are described in figure 3.3. Note however that these extensions are very experimental yet

A guard can now be the succession of several atomic events, possibly optional or on the contrary repeated more than one time. The repetition modifier follows the syntax and semantics of POSIX regexps: the most general are {e1,e2} that indicates at least e1 repetitions and at most e2 and {e1,} that indicates at least e1 repetitions without upper bound. There are then shortcuts for the most common patterns:

- no modifier indicates exactly one execution (equivalent to {1,1})
- + indicates 1 or more repetitions (equivalent to {1,})
- * indicates any number of repetitions, including 0 (equivalent to {0,})
- ? is equivalent to {0,1}
- {e} is equivalent to {e,e}
- {,e} is equivalent to {0,e}

Note that a repetition modifier that allows a non-fixed number of repetitions prevents the automaton to be **%deterministic**.

id(seq) indicates that we have a **CALL**(id) event, followed by the internal sequence of event, and a **RETURN**(id), *i.e.* it describes a complete call to id, including the calls that id itself performs. In particular, f() indicates that f does not perform any call. When in a sequence internal to a call to f, the identifiers found in the expressions are first searched among the formals of f, starting with the innermost call and then among globals. It is still possible to use f().x to refer to parameter x of f, but if f is already in the call stack, this will not trigger a new **CALL**(f) event at this point. Instead, the value of x for the last call to f will be used.

In addition, the **CALL**(id) event may be further guarded by a pre-condition, that is either the name of an ACSL behavior of id, or a basic YA condition (in which we have access to the formals of id as explained above). Similarly, the final **RETURN**(id) event can come with a post-condition, in which one can access the **\result** returned by id.

```
guard          ::=  seq-elt

seq-elt        ::=  condition | basic-elt repetition

basic-elt      ::=  [ non-empty-seq ] | id pre-cond ( seq ) post-cond

seq            ::=  ϵ | non-empty-seq

non-empty-seq  ::=  seq-elt | seq-elt ; seq

repetition     ::=  ϵ | + | * | ?
                  | { expr , expr } | { expr }
                  | { expr , } | { , expr }

pre-cond       ::=  ϵ | :: id | {{ condition }}

post-cond      ::=  ϵ | {{ condition }}
```

Figure 3.3: Extended YA guards

For instance, the following automaton describes a function main that does not call anything when called in behavior `bhv` and performs a single call to `f`, when called with a parameter `c` less than or equal to `0`, returning `0` in this latter case:

```
%init: S0;
%accept: Sf;

S0: { main::bhv() } -> Sf
  | { main {{ c <= 0 }} (f()) {{ \result == 0 }} } -> Sf;

Sf: -> Sf;
```

YA variables

Extended guards do not allow specifying relations between the parameters of distinct, non-nested calls. In order to be more flexible, it is possible to declare variables in a Ya file, to assign them values when crossing a transition, and to use them in guards. The syntax for that is described in Fig. 3.4.

Only `char`, `int`, `long` variables are currently supported. Furthermore, variables can only be introduced in deterministic automata, which do not use extended guards.

A variable `$x` must have been declared in the directives of the file to be used in a guard. Furthermore if it is used in a transition starting from state `S`, then all possible paths from the initial state to `S` must contain at least one assignment to `$x`. Note that assignments are performed sequentially, so that if `$x` has already been assigned in a given sequence of actions, it can automatically be used in subsequent assignments (on the other hand, since conditions are evaluated before actions, it must have been initialized elsewhere if it were to be used in the condition part of the guard.

directive ::= ... | `$` *id* `:` *type*

type ::= `char` | `int` | `long`

guard ::= `{` *condition* `}` *action**

action ::= `$` *id* `:=` *lval* `;`

lval ::= ... | `$` *id*

Figure 3.4: Syntax for declaring and using YA variables

in the right hand side of an action, it is possible to refer to the value of a formal parameter of `f` when the transition is triggered over a **CALL** to `f` and to its return value when hangling a **RETURN** event, as described in section 3.1.2 for conditions.

An example of YA automaton with variables is given below. It uses variables `$x` and `$y` that are updated when calling `f` and returning from `i`, while `$x` is used when calling `h`.

```
%init:      a;
%accept:    i;
%deterministic;

$x : int;
$y : int;

a : { CALL(main) } -> b;

b :
  { CALL(f) } $x:=f().x; $y := $x; -> c
| { CALL(g) } -> d
;

c : { RETURN(f) } -> e;

d : { RETURN(g) } -> g;

e :
  { CALL(h) && $x > 0 } -> f
| { RETURN(main) } -> i
;

f : { RETURN(h) } -> g;

g : { CALL(i) } -> h;

h : { RETURN(i) } $y := 0; $x := 1; -> e;

i : -> i;
```

Advanced Features 4

4.1 Generated Annotated Program

The instrumented program is the original program (with its annotations[1]) completed with the following:

- Some auxiliary C declarations representing the automaton itself and information needed to decide if a given transition should be taken or not;
- If the automaton has been marked as `deterministic`, a set of lemmas state that it is indeed the case;
- For each original C function, two functions are defined with their specification. They take care of updating the automaton's state when entering and exiting the function respectively;
- Each original C function gets additional ACSL behaviors, expressing how the automaton is supposed to evolve when the function is called
- Each loop gets additional loop invariants stating in which states the automaton might be during the loop.

These annotations are detailed in the rest of this section.

4.1.1 Auxiliary Variables

We have to represent the current state of the automaton. It can take two forms. First, if the automaton is marked as **`%deterministic`**, an **`enum`** type representing the states of the automaton is generated. It makes it easier to read the generated annotations when they come from a Ya file with explicitly named states. We use then a single variable, `aorai_CurStates` which is simply a value of the **`enum`** type corresponding to the current (unique) active state of the automaton. Otherwise, we use a set of boolean variables, whose value is `1` when the automaton is in the corresponding state and `0` otherwise.

Furthermore, the use of extended YA constructions (section 3.1.3) might introduce additional variables:

- Repetitions introduce a counter, `aorai_counter` (with a numeric suffix if needed), except if their lower bound is `0` or `1` and they don't have an upper bound or their upper bound is `0` or `1` (in these cases, there is no need to test the number of repetition done so far at the end of the sequence).
- The value of a parameter `prm` of function `f` that is accessed in another event than `CALL(f)` is stored in a global variable `aorai_prm` in order to be accessible in the remainder of the sequence.

4.1.2 Deterministic lemmas

When a YA automaton is marked as **`%deterministic`**, some lemmas are generated whose verification will ensure that the automaton is indeed deterministic. Namely, for each state of the automaton, a

[1] ACSL language for annotation is described at https://github.com/acsl-language/acsl

lemma states that at any given event, there is at most one transition exiting from this state that is active.

4.1.3 Update functions

In order to update the automaton's status, a pair of function is defined for each function `f` defined in the original C code. `f_pre_func` is then called when entering `f`, while `f_post_func` is called just before `f` returns. Both come with a specification that indicates what actions may occur for the automaton at the corresponding event. For instance, we can have a look at the specification generated for `opa_pre_func` in our running example, presented in figure 4.1. Similarly, the corresponding body is shown in figure 4.2. For each state of the automaton, we have one or two behaviors, describing

```
/*@ ensures aorai_CurOpStatus == aorai_Called;
    ensures aorai_CurOperation == op_opa;
    assigns aorai_CurOpStatus, aorai_CurOperation,
            S1, S2, S3, S4, S5, S6, S7;

    behavior buch_state_S1_out:
      ensures 0 == S1;

    behavior buch_state_S2_out:
      ensures 0 == S2;

    behavior buch_state_S3_in:
      assumes 1 == S2 && r >= 0;
      ensures 1 == S3;

    behavior buch_state_S3_out:
      assumes 0 == S2 || !(r >= 0);
      ensures 0 == S3;

    behavior buch_state_S4_out:
      ensures 0 == S4;

    behavior buch_state_S5_out:
      ensures 0 == S5;

    behavior buch_state_S6_out:
      ensures 0 == S6;

    behavior buch_state_S7_out:
      ensures 0 == S7;
 */
void opa_pre_func(int r);
```

Figure 4.1: Specification of `opa_pre_func`

whether the state can be active or not. In addition, when there are counters or other auxiliary variables that must be updated, other `ensures` clauses define their new value according to the transition that is activated.

It is also possible to only activate the generation of the body of the transition functions, without their specification (e.g. to analyze the instrumented code with the Eva plug-in, which does not need the con-

```
int S1_tmp;
int S2_tmp;
int S3_tmp;
int S4_tmp;
int S5_tmp;
int S6_tmp;
int S7_tmp;
aorai_CurOpStatus = aorai_Called;
aorai_CurOperation = op_opa;
S1_tmp = S1;
S2_tmp = S2;
S3_tmp = S3;
S4_tmp = S4;
S5_tmp = S5;
S6_tmp = S6;
S7_tmp = S7;
S7_tmp = 0;
S6_tmp = 0;
S5_tmp = 0;
S4_tmp = 0;
if (S2 == 1 && r >= 0)
  S3_tmp = 1;
else S3_tmp = 0;
S2_tmp = 0;
S1_tmp = 0;
S1 = S1_tmp;
S2 = S2_tmp;
S3 = S3_tmp;
S4 = S4_tmp;
S5 = S5_tmp;
S6 = S6_tmp;
S7 = S7_tmp;
```

Figure 4.2: Body of `opa_pre_func`

tracts and loop invariants). This is done through option `-aorai-no-generate-annotations`. In that case, it might be the case that the automaton end up in a rejecting state (for a deterministic automaton) or without any active state (for a non-deterministic automaton). Option `-aorai-smoke-tests` can thus be used to generate assertions at the end of all update functions, stating that the automaton is still in an appropriate state.

4.1.4 Functions behaviors

Each function `f` defined in the original C code gets its specification augmented with behaviors describing how the automaton's status changes during a call to `f`. The specification of the `opa` function in our running example is shown in figure 4.3.

```
/*@ requires
      1 == S2 &&
      0 == S1 && 0 == S3 && 0 == S4 &&
      0 == S5 && 0 == S6 && 0 == S7;
    requires 1 == S2 ==> r >= 0;
    requires r < 5000;

    behavior j:
      ensures \result == \old(r)+1;

    behavior Buchi_property_behavior:
      ensures 1 == S4 ==> \result <= 5000;
      ensures 0 == S1 && 0 == S2 && 0 == S3 &&
              0 == S5 && 0 == S6 && 0 == S7;
      ensures 1 == S4;
 */
int opa(int r);
```

Figure 4.3: Generated specification for an existing C function

The first **`requires`** clause indicates which state(s) can be active before entering the function. Then, for each of these states, we have a requirement that at least one of the guard of a transition exiting from this state is true.

After the global **`requires`**, we find some **`behavior`**s corresponding to the possible states of the automaton when the function returns.

Again, we might also find some post-conditions on the auxiliary variables used by Aoraï. Note however that these conditions are computed through abstract interpretation and may thus be over-approximated.

4.1.5 Loop Invariants

For each loop, Aoraï defines an invariant stating in which states the automaton can be during the loop. Since the states of the automaton when entering the loop the first time and the states found during the executions of the loop can be quite different, Aoraï introduces in addition a new variable, that is initially set to 1 and reset to 0 when the loop is entered. This allows to make a distinction between the first run and the other ones and to refine the invariant according to value of the variable. Possible values for the auxiliary variables are also described by loop invariants (again, the values found might be over-approximated).

An example of loop invariant can be found using the following example. Figures 4.4 and 4.5 describe the automaton and the C code (a function `main` is supposed to call `f` and `g` between 0 and 5 times). Figure 4.6 presents the generated invariants for the **`while`** loop.

```
%init: S0;
%accept: Sf;

S0: { [main([f();g()]{0,5})] } -> Sf;
Sf: -> Sf;
```

Figure 4.4: Example of YA automaton describing a loop

```
int f() {}

int g() {}

int main(int c) {
  if (c<0) { c = 0; }
  if (c>5) { c = 5; }
  /*@ assert 0<=c<=5; */
  while (c) {
    f();
    g();
    c--;
  }
  return 0;
}
```

Figure 4.5: Original C code with a loop

4.2 Analyzing Annotated Files with Eva or WP

Once the annotated file has been generated, it remains to verify that all the annotations hold. This section describes briefly how this can be done and some common issues that may arise during verification.

In addition to annotations, Aoraï generates an implementation for the transition functions, so that it is possible to use the Eva plug-in of Frama-C on the instrumented code. However, there's no guarantee that Eva will be able to perform an analysis that stays precise enough to verify that the automaton always ends in an accepting state. Aoraï will set up automatically a certain number of parameters in Eva to help make the analysis more precise, but, in the case of deterministic automata it is also possible to use option `-aorai-instrumentation-history n` to have the instrumentation retain the `n` previous states of the automaton (in addition to the current state), that will be used for splitting Eva's abstract states. Furthermore, each time Eva encounters a call to the built-in function `Frama_C_show_aorai_state`, it will display the current state (together with the previous ones up to `-aoraiinstrumentation-history`). If the function is called with some arguments, their abstract value will also be displayed.

Another possibility is to use the deductive verification plug-in WP. Note however that the generated annotations are not guaranteed to be complete, *i.e.* to it might be necessary to add further annotations in order to discharge all proof obligations. In particular, in presence of loops, Aoraï generates loop invariants for its own auxiliary variables, but it is likely that these variables (especially the counters) will need to be related to the variables of the original programs. For instance, we must add to the loop invariants of figure 4.6 that `c+aorai_counter` remains constant throughout the loop (`c` gets decremented at each step, while `aorai_counter` gets incremented), but such a relation is well beyond the scope of Aoraï itself.

```
/*@ loop invariant Aorai: 0 == S0;
    loop invariant Aorai: 0 == Sf;
    loop invariant
     Aorai:
       1 == aorai_intermediate_state ||
       0 == aorai_intermediate_state;
    loop invariant
     Aorai:
       1 == aorai_intermediate_state_0 ||
       0 == aorai_intermediate_state_0;
    loop invariant Aorai: 0 == aorai_intermediate_state_1;
    loop invariant Aorai: 0 == aorai_intermediate_state_2;
    loop invariant Aorai: 0 == aorai_intermediate_state_3;
    loop invariant
     Aorai:
       1 == aorai_intermediate_state ||
       1 == aorai_intermediate_state_0;
    loop invariant
     Aorai:
       aorai_Loop_Init_43 != 0 ==>
       \at(1 == S0,Pre) ==>
       0 == aorai_intermediate_state_0;
    loop invariant
     Aorai: aorai_Loop_Init_43 == 0 ==>
          0 == aorai_intermediate_state;
    loop invariant
     Aorai:
       \at(1 == aorai_intermediate_state,aorai_loop_43) &&
       1 == aorai_intermediate_state_0 ==>
       1 <= aorai_counter <= 5;
 */
```

Figure 4.6: Example of Generated Loop Invariants

Going Further 5

The objective of the Aoraï plug-in is to generate an annotated C program such that, if it is validated, then the original program respect the automaton. In this chapter we first introduce some theoretical bases on the approach by annotation generation. Next we describe the two parts of the computing module:

- the specification generator (from the automaton)
- the constraints propagation for static simplification.

5.1 Theoretical Base of the Approach

A program can be defined by a set of execution traces $PATH_{Prog}$ and similarly, an automaton can be defined by a set of accepted traces $PATH_{\text{Büchi}}$. Hence, to verify that a program is correct with respect to an automaton, we need to verify two aspects:

Safety for each program trace t, there exists a Büchi path c, such that, for each i, the cross-condition P_i from the c is verified in the context of the state t_i (figure 5.1). More formally, we have:

$$\forall t \in PATH_{Prog} \cdot \exists c \in PATH_{\text{Büchi}} \cdot \forall i \in 0..(size(t)-1) \cdot t_i \models P_i(c)$$

Liveness for each program trace t, there is an infinite number of states synchronized with a Büchi acceptance state. We propose to restrict these constraints to the weaker one : there is no deadlock (always a crossable transition from a non-acceptance state) and no live-lock (always a finite number of states between 2 acceptance states).
Note: At this time the liveness aspect is not included in the tool.

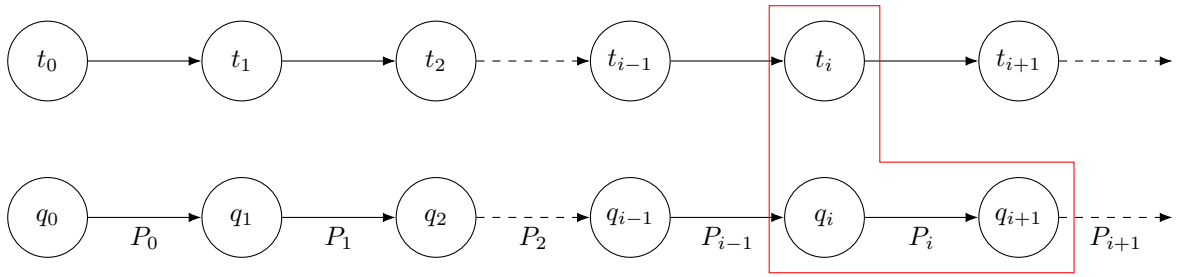


Figure 5.1: Synchronization of Paths from automata and from Program

5.1.1 Safety

In order to encode this approach in an approach by annotations and to consider all program traces, our solution is to use a synchronization function. Such a function associates the set of states synchronized

with the n$^{\text{th}}$ state from an execution trace. It is the sufficient to prove that at least one state is synchronized with each state of the execution to establish the safety of the property.

Definition 1 (Synchronization function)
Let $A = \langle Q, q_0, R\rangle \in BUCHI$ *and* $\sigma \in PATH_{Prog}$. *The synchronization function* $Sync \in BUCHI \times PATH \times \mathbb{N} \to 2^Q$ *is defined by:*

- $Sync(A, \sigma, 0) = \{q_0\}$
- *For each* $i > 0$*:*

$$Sync(A, \sigma, i) = \left\{ q' \;\middle|\; \begin{array}{c} \exists \langle q, P, q'\rangle \in R \cdot \wedge \\ \sigma_{i-1} \models P \wedge \\ q \in SyncA, \sigma, i-1) \end{array} \right\}$$

Definition 2 (Acceptance condition)
(C_{Sync}) $\qquad \forall i \in 0..(len(\sigma) - 1) \cdot Sync(A, \sigma, i) \neq \emptyset$

This verification is encoded into annotations by generating the following assertions:

Declaration Let $\{q_0, \ldots, q_n\}$ be a set of boolean variables associated to the states. q_i is true if the system is synchronized with the state i. Initially, only q_0 is true.

Transitions A set of ghost instructions has to be generated just before each call and return statement. These instructions have to update the set of states synchronized with the current state.

Synchronization The synchronization condition can be expressed with an invariant verifying that at least one state is always synchronized.

5.1.2 Liveness

This part is not developed at this time, but the method consists in verifying a global variant between each couple of acceptance states and also the inclusion of the set of reachable states in the set of accepting states.

5.2 Adding from the Theory

The previous section described a sufficient framework. However, in order to verify the correction with theorem provers, we need to use more efficient modeling and to add some hypothesis in order to link the models from C program and the automaton.

5.2.1 Automata Modeling

In order to link models from the program and the property, we describe the automaton as constants in the generated C file. This axiomatization is combined with a set of invariants that give some properties of the automaton. For instance, the non-reachability of a state s can be deduced from the absence of transitions from an active state to s such that its cross-condition is true. This cross-condition is then expressed in terms of program information. This is the link program-automata.

5.2.2 Memorization of last Transitions

In order to memorize the last synchronization link, we keep the set of last crossed transitions in addition with the set of old active states.

5.2.3 Use of Specifications instead of Invariant

Finally, the synchronization condition is not implemented as an invariant, but as a pre- and post-condition on each operation. This choice is more flexible if we can statically decide that some states cannot be

synchronized with some operation. In the following section, our objective is to describe how to automate this simplification by using abstract interpretation.

5.3 Abstract Interpretation

Current Implementation : behavioral Property as Widening Operator

In this section we describe our method to generate the specification of each operation. In a first part, we deduce an over-approximation of specifications by using automata, and next we propagate the generated constraints in order to converge to a fixpoint of specifications.

5.3.1 Generation of Abstract Specifications

Initially, each operation's specification states that each state and transition can be active before and after an operation. We then fix a first constraint: the main operation starts in the initial state. Next, we verify, for each operation, if its call or its return is always forbidden in a particular transition's cross-condition. If any, the associated transition is removed from the operation's specification. This process is done once on each operation. Finally, this computed constraint has to be propagated.

5.3.2 Static Simplification

Starting from specified operations, each of them is analyzed by forward and backward abstract interpretation. The abstraction consists in abstracting all expressions. We only consider control statements and call and return statements.

The post-condition is defined by intersecting its old value with the reachable post-condition computed by forward propagation. Similarly, the pre-condition is defined by intersecting its old value with the reachable pre-condition computed by backward propagation.

If a loop is reached during this process, we compute its loop invariant in terms of automata from its computed pre- and post-conditions.

During each pass of the program the list of use-cases of each operation is kept. Hence, if we observe that an operation is still called from a strict subset of its authorized input states, then we restrict its specification.

Finally, a fixpoint is computed in order to minimize the specifications.

Note that during this process, the post-conditions are described as behaviors. Indeed, this approach allow to give a particular post-condition for each possible pre-condition. Hence, the caller, which cannot observe the control-flow inside a called operation, has more precise information about current active states, since it knows each previous active states.

5.4 Plugin Architecture

The plug-in is composed of three parts:

1. a front-end (translator);
2. a computing module for specification of operations;
3. a back-end (C generator, including annotations).

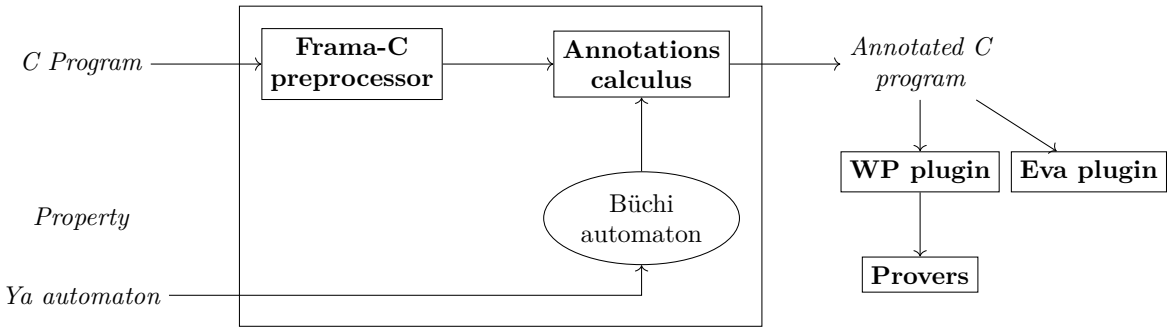


Figure 5.2: Plug-in Structure

5.5 Recent updates

5.5.1 Frama-C Iron

- Remove obsolete support for LTL and Promela input language

5.5.2 Frama-C Vanadium

- Documentation for options `-aorai-no-generate-annotations` and `-aorai-smoke-tests`
- Documentation for option `-aorai-instrumentation-history` and built-in `Frama_C_show_aorai_state`
- Aoraï does not generate a C file by default anymore, relying on kernel options `-print` and `-ocode` for that, like all plug-ins. Remove corresponding ad'hoc options.
- update syntax for YA sequence to avoid ambiguities with `+` and `*` repetition operators

5.5.3 Frama-C Titanium

- Various bug fixes
- Introduction of YA variables

5.5.4 Frama-C Aluminium

- Generated functions now have a body in addition to a specification

5.5.5 Frama-C Nitrogen

- New translation mechanism for the automaton
- Extended Ya guards

5.5.6 Frama-C Boron

- A function that is used in a C program, but that is not defined is stubbed by Frama-C and ignored in Aorai.
- For each function and each loop, if no state can be enabled before or after it (not reachable), then a warning is displayed. It is usually either a dead code, or a code violating the specification.
- In the YA and Promela formats, it is now possible to speak about call parameters and returned value. `f().a` denotes the call parameter `a` of `f` and `f().return` denotes the returned value of f.

– In the annotated C file generated, array of states are indexed by the name of the state (defined as an enum structure)

5.5.7 Frama-C Beryllium

– YA format for properties

Conclusion 6

This manual is not always up-to-date and only gives some hints on the Aoraï plug-in. If you want more information, please send me a mail at:

nicolas.stouls@insa-lyon.fr

or visit the website:

https://www.frama-c.com/fc-plugins/aorai.html