Aoraï Plugin Tutorial

(A.k.a. LTL to ACSL)

Nicolas Stouls
Nicolas.Stouls@insa-lyon.fr

April 3, 2009

Foreword

The Aoraï plugin provides a method to automatically annotate a C program according to an LTL formula F such that, if the annotations are verified, then we ensure that the program respects F.

The classical method to validate annotations is to use the Jessie plugin and the Why tool.

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

Contents

Chapter 1

Plugin Quick Start Installation

Classically, from Frama-C sources, the `configure` command returns following information about Aoraï plugin:

```
(...)
checking for src/ltl_to_acsl... yes
ltl_to_acsl... yes
configure: ******************************************************
configure: * CONFIGURE TOOLS AND LIBRARIES USED BY SOME PLUGINS *
configure: ******************************************************
checking for ltl2ba... no
configure: WARNING: ltl2ba not found.
plugins disabled:
  ltl_to_acsl
(...)
configure: ltl to acsl          : no (see warning about ltl2ba)
```

You then need to install[1] the ltl2ba tool in your current path. Next, re-run the `configure` command and check that you have the following lines:

```
configure: ******************************************************
configure: * CONFIGURE TOOLS AND LIBRARIES USED BY SOME PLUGINS *
configure: ******************************************************
checking for ltl2ba... yes
(...)
configure: ltl to acsl          : yes
```

Finally, just do a `make`/`sudo make install` and enjoy. In case of problems, please refer to the Frama-C manual.

[1] From `http://www.lsv.ens-cachan.fr/~gastin/ltl2ba/index.php`

Chapter 2

Introduction

2.1 Concrete Example

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

/*@ requires rr<5000;
  @ behavior j :
  @ ensures rr==3;
*/
void opa() {rr++;}

void opb () {rr+=2;}

void opc () {rr=60000;}
```

```
int main(){
     if (rr<5000) goto L;
     opc();
  L4:goto L5;
  L :opa();
     goto L2;
     opc();
  L6:return 1;
  L3:goto L4;
     opc();
     goto L2;
  L2:goto L3;
  L5:opb();
     goto L6;
}
```

Figure 2.1: Example of C File

```
CALL(main) && _X_ (CALL(opa) && _X_ (!RETURN(opb) && _X_
(!CALL(opa) && _X_ (RETURN(opb) && _X_ (RETURN(main))))))
```

Figure 2.2: Example of LTL Formula

The call of Aoraï is done through `frama-c prog.c -ltl formula.ltl`. In a first step, the LTL formula is syntactically simplified and sent to the *LTL2BA* tool, which generates an associated Büchi automata (Fig. 2.3).

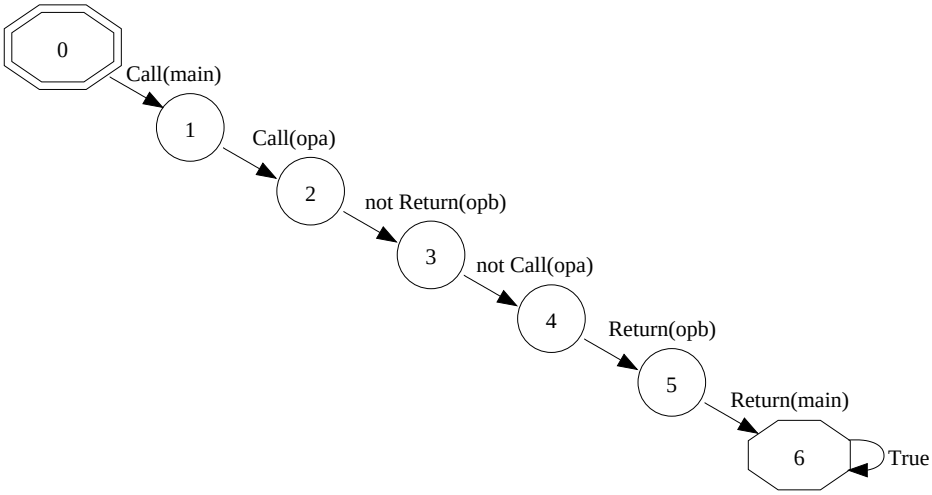

Figure 2.3: Büchi automata from the given LTL Formula

Next, a specification is computed for each operation, in terms of states and transitions from the Büchi automata. For instance, the previous example leads to the following specification :

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

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

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

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

Finally, Aoraï generates a new C program, including the Büchi automata axiomatization, some coherence invariants and annotations on operations, such that if this annotated program can be validated with the Jessie plugin, then we ensure that it respects the given LTL formula.

This whole example and its resulting file can be found in:

`http://www.frama-c.cea.fr/download/ltl_to_acsl/example.tgz`

2.2 General View of Aoraï

The Aoraï plug-in is composed of three parts:

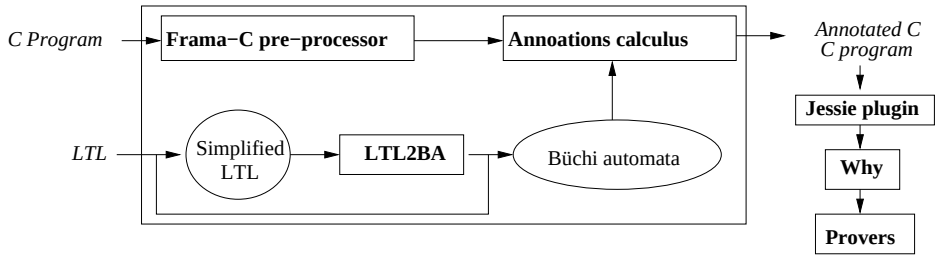

Figure 2.4: Plug-in Structure

1. a front-end (LTL to Büchi translator, based on the *LTL2BA* tool);
2. a computing module for specification of operations;
3. a back-end (C generator, including annotations).

Their interaction is described figure 2.4. The computing module is the main part of the plug-in. Its description is in chapter 4. The Front end is described in chapter 3 and the back-end is not described.

2.3 Known restrictions

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

- Only the safety part of the LTL formula is check. The liveness part is not consider. The first version of the theory is developed in the J. Groslambert PhD thesis and in an implementation way, acceptance states are starting to be managed, but not variants.
- Currently, the `switch` and `unordered` statements are not supported.

Chapter 3

Inputs/Outputs

The Aoraï plug-in needs 2 files (a C program and a LTL formula) and generates a C program including annotations. In the following section, I present the syntax of the LTL file and in the following section I describe Aoraï inline options.

3.1 Syntax of the LTL Used

The property to verify has to be described in LTL logic, in a `.ltl` file. Figure 3.1 gives the general syntax of the supported LTL. The ASCII representation of these operators is, as much as possible, the one of the C language. Particular cases are described fig. 3.2. Syntax of modalities is inspired from the one of the *LTL2BA* tool (which is used to translate LTL formula in Büchi automata). However, in order to suppress some constraints on input language (such as no expressions or uppercase variables), we prefix and postfix each *LTL2BA* modality with an underscore.

/ Formula */*
F ::=
(1^{st} order) TRUE | FALSE | '(' F ')' | $F \vee F$ | $F \wedge F$ | $\neg F$ | $F \Rightarrow F$ | $F \Leftrightarrow F$
(LTL) | '$\Box$' F | '$\Diamond$' F | F 'UNTIL' F | F 'RELEASE' F | 'NEXT' F
(Predicates) | 'CALL'(Ident) | 'RETURN'(Ident) | 'CALL_OR_RETURN'(Ident)
(Exprs) | E

/ Expressions */*
E::= R '=' R | R '<' R | R '>' R | R '$\leq$' R | R '$\geq$' R | R '$\neq$' R | R
R::= R '+' R | R '-' R | R '*' R | R '/' R | R '%' R | A
A::= Int | (R) | $Ident$('['R']')$^+$ | Ident

Figure 3.1: Grammar of the LTL Logic Used

LTL Operators	ASCII	LTL Operators	ASCII
TRUE	`true`	$\Box$	`_G_`
FALSE	`false`	$\Diamond$	`_F_`
$\Rightarrow$	`=>`	UNTIL	`_U_`
$\Leftrightarrow$	`<=>`	RELEASE	`_R_`
		NEXT	`_X_`

LTL Operators	ASCII
CALL	`CALL`
RETURN	`RETURN`
CALL_OR_RETURN	`CALL_OR_RETURN`

Figure 3.2: ASCII Syntax of the LTL Logic Used

Finally, figure 3.3 is a concrete example of a LTL formula and its ASCII description. In this manual, we will prefer the mathematical notation.

Atomicity Property	
(Natural)	b is called only if a is called immediately before and did not return an error.
(LTL)	$\Box((\neg\mathbf{RETURN}(a) \ \vee \ \neg status) \Rightarrow \bigcirc\neg\mathbf{CALL}(b))$
(ASCII)	`_G_((!RETURN(a)) || !status) => _X_!CALL(b))`

Figure 3.3: Concrete example of LTL formula

3.2 Aoraï Usages

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

-ltl <s> Where <s> is the location of the file containing the LTL property

-ltl-verbose Gives some information during computation, such as used/produced files and heuristics applied

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

-ltl-dot Generates a dot file of the Büchi automata. Dot is a graph format used by the GraphViz tool[1].

Finally, here is a concrete example of a common call:

```
frama-c prog.c -ltl formula.ltl -show-op-spec
```

[1] `http://www.graphviz.org`

3.3 Generated Annotated File

The default configuration is to generate a new C file with the same name as the original program and suffixed by `_annot` (If the file already exists, and numeric suffix is added). The generated file is the original program (with its annotations) completed with 6 types of information:

- An axiomatization of Büchi automata associated to the property (Sect. 3.3.1);
- Some variables modellizing the current states and transitions of the Büchi automata (Sect. 3.3.2);
- Some invariants characterizing links between program specification and Büchi automata (Sect. 3.3.3);
- Additional pre and post-conditions for each operation, in terms of the states and transitions of the Büchi automata (Sect. 3.3.4);
- Some piece of ghost code before each call and each return statement, which updates the current state of the Büchi automata (Sect. 3.3.5);
- Loop invariants in terms of the Büchi automata (Sect. 3.3.6).

For each if these information we give (figure 3.4 to 3.8) a piece of the C file generated according to the example from section 2.1.

3.3.1 Büchi Automata Axiomatization

The automata is a set of transitions and each transition is a triplet of a starting state, a stopping state and a cross-condition. Our axiomatized representation is composed of :

- 2 logic functions that associate, to a transition number, its starting or endding state
- a predicate *(parameterized by a transition number, the current operation and its status)* which is true if and only if the associated cross-condition is true

An example is given figure 3.4.

3.3.2 Variables

Three variables are generated. They respectively modellize the set of possible current states, the set of possible passed over transitions and the set of last active states. These variables are described by tables of int, where each cell is a state (resp. a transition). If a cell is zero then the state/transition is not active. The initial state of these variables corresponds to the call of the main. Hence, the initial state from the Büchi automata is active in the last states and the current active transitions are the one with a condition which accepts call(main). Current states are the endding states of these transitions. An example is given figure 3.5.

```
/*@ axiomatic transStart {
  @   logic integer transStart(integer tr) ;
  @   axiom transStart0: (transStart(0) == 0);
  @   axiom transStart1: (transStart(1) == 1);
  @   ...}
*/
/*@ axiomatic transStop {
  @   logic integer transStop(integer tr) ;
  @   axiom transStop0: (transStop(0) == 1);
  @   axiom transStop1: (transStop(1) == 2);
  @   ...}
*/
/*@ predicate transCond{L}(integer numTr, integer op, integer status) =
  @     (numTr == 0 ⇒ op == op_main   ∧   status == Called)
  @   ∧ (numTr == 1 ⇒ op == op_opa   ∧   status == Called)
  @   ∧  ...
*/
```

Figure 3.4: Example of Büchi Automata Axiomatization

```
int curSt[7] = {0, 1, 0, 0, 0, 0, 0};
int curTr[7] = {1, 0, 0, 0, 0, 0, 0};
int buch_CurStates_old[7] = {1, 0, 0, 0, 0, 0, 0};
```

Figure 3.5: Example of Generated Variables

3.3.3 Invariants

Some invariants are used to join model variables and to link the specifications of the Büchi automata and of the program. For instance, the invariant given figure 3.6 is a condition sufficient to establish that a state is not active. This invariant depends on the *transCond* predicate which is express in terms of the program variables.

3.3.4 Specifications

Generated specifications describe current states and transitions. Each pre and post condition is composed of 4 assertions.

- Set of impossible transitions;
- Set of possible transitions;
- Set of non-active states;
- Set of active states.

/*@ ***global invariant*** Unreachability_1:
@ $\forall st; 0 \leq st < NbStates \ \wedge$
@ $\left(\begin{array}{l} \forall tr; 0 \leq tr < NbTrans \\ \quad \Rightarrow curTr[tr] = 0 \ \vee \ transStop(tr) \neq st \ \vee \\ \quad\quad \neg transCond(tr) \ \vee \ buch\_CurStates\_old[transStart(tr)] = 0 \end{array}\right)$
@ $\Rightarrow curSt[st] = 0;$
*/

Figure 3.6: Example of Generated Invariant

In order to be more precise, postconditions are described in terms of input states. Hence, there is one behavior for each possible active state in precondition, such as described in figure 3.7.

requires $0 == curTr[0] \wedge 0 == curTr[2] \wedge 0 == curTr[3] \wedge 0 == curTr[4] \wedge 0 == curTr[5] \wedge 0 == curTr[6]$
requires $0 != curTr[1]$
requires $0 == curSt[0] \wedge 0 == curSt[1] \wedge 0 == curSt[3] \wedge 0 == curSt[4] \wedge 0 == curSt[5] \wedge 0 == curSt[6]$
requires $0 != curSt[2]$
behavior buch_0:
assumes $0 != curSt[2]$
ensures $0 == curTr[0] \wedge 0 == curTr[1] \wedge 0 == curTr[3] \wedge 0 == curTr[4] \wedge 0 == curTr[5] \wedge 0 == curTr[6]$
ensures $0 != curTr[2]$
ensures $0 == curSt[0] \wedge 0 == curSt[1] \wedge 0 == curSt[2] \wedge 0 == curSt[4] \wedge 0 == curSt[5] \wedge 0 == curSt[6]$
ensures $0 != curSt[3]$

Figure 3.7: Example of Generated Specifications for *opa*

3.3.5 Synchronization Code

Before each call of operation and before each return statement, a piece of code is introduce in order to update the current status of the Büchi automata. Each of them is composed of 4 parts:

- Update of the current operation and of its status;
- Backup of current active states into the old states;
- Computation of new active states;
- Computation of transitions that are crossed.

Note than, since cross conditions are statically simplified, the described conditions can be slightly difficult to match with the Büchi cross conditions. Figure 3.8 gives a concrete example of such a synchronization code.

```
{Operation= op_ opa;
 Status= buch_ Terminated;
 buch_ CurStates_ old[1] = curSt[1];
 buch_ CurStates_ old[2] = curSt[2];
 ...
 curSt[0] = 0;
 ...
 curSt[3] = buch_ CurStates_ old[2];
 ...
 curTr[0] = 0;
 curTr[1] = 0;
 curTr[2] = buch_ CurStates_ old[2];
 ...
 return;
}
```

Figure 3.8: Example of Generated Synchronization Code

3.3.6 Loop Invariants

Each loop as to be specified in terms of the automata states and transitions. The generated invariant has then the same structure as the generated pre/post conditions, with 4 parts. However, we introduce a subtlety in order to dissociate the first iteration and the others. A fresh variables is then introduce and used to separate these cases. An example is given figure 3.9

```
/*@ loop invariant
  @        (0 != curSt[0]  ∨  0 != curSt[1])  ∧
  @        true  ∧
  @        (0 != curTr[1]  ∨  0 != curTr[2]  ∨  0 != curTr[3])  ∧
  @        0 == curTr[0];
  @ loop invariant buch_ Loop_ Init_ 23 != 0 ⇒
  @        curSt[0] == 0  ∧  curTr[2] == 0  ∧  curTr[3] == 0;
  @ loop invariant buch_ Loop_ Init_ 23 == 0 ⇒
  @        curTr[1] == 0;
*/
```

Figure 3.9: Example of Generated Loop Invariants

Chapter 4

Verifying LTL Formula

The objective of the Aoraï plug-in is to generates an annotated C program such that, if it is validated, then the original program respect the LTL property. 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 LTL property)
- the constraints propagation for static simplification.

4.1 Theoretical Base of the Approach

A program can be defined by a set a execution traces $PATH_{Prog}$ and similarly, an LTL formula can be defined by a set of accepted traces $PATH_{\text{Büchi}}$. Hence, to verify that a program is correct with respect to a LTL formula, 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 t_i state (Figure 4.1). 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 infinity number of states synchronized with a Büchi acceptance state. We propose to restrict this constraints to the weaker one : there is no dead-lock (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 include in the tool.

4.1.1 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

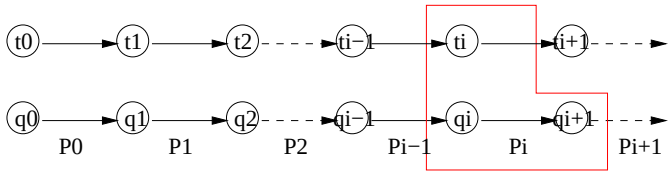

Figure 4.1: Synchronization of Pathes from Büchi and from Program

function associates the set of Büchi states synchronized with the n^{th} state from an execution trace. It is the sufficient to prove that at least 1 Büchi 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* $Sync \in BUCHI \times PATH \times \mathbb{N} \rightarrow 2^Q$ *is defined with:*

- $Sync(A, \sigma, 0) = \{q_0\}$
- *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 encode into annotations by generating following assertions:

Declaration let $\{q_0, \ldots, q_n\}$ a set of boolean variables associated to the Büchi states. q_i is true iff the system is synchronized with the Büchi 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 has to update the set of Büchi states synchronized with the current state.

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

4.1.2 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 the inclusion of the reachable states into the acceptance states set.

4.2 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 modellization and to add some hypothesis in order the link the models from C program and LTL property.

4.2.1 Büchi Automata Modellization

In order to link models from the program and the property, we describe the Büchi automata as constants in the generated C file. This axiomatization is combined with a set of invariant that gives some property to the automata. For instance, the non-reachability of a state s can be deduce from the non existence of transition from an active state to s such that its cross condition be true. This cross condition, is then expressed in terms of program information. This is the link program-Büchi.

4.2.2 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.

4.2.3 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 can not by synchronized with some operation. In the following section, our objective is to described how to automate this simplification by using abstract interpretation.

4.3 Abstract Interpretation.

Current Implementation : LTL 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 Büchi automata, and next we propagates the generated constraints in order to converge into a fix-point of specifications.

4.3.1 Generation of Abstract Specifications

Initially, each operation specification is 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 allways forbidden in a particular transition cross condition. If any, the associated transition is removed from the operation specification. This

process is done once on each operation. Finally, this computed constraints has to be propagated.

4.3.2 Static Simplification

Starting from specified operations, each of them is analyzed by froward 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 reach during this process then we compute its loop invariant in terms of Büchi automata from its computed pre and post conditions.

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

Finally, a fixpoint is computed in order to minimize 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 can not observe the control flow inside a called operation, have more precise information about current active states, since it knows each previous active states.

4.4 Conclusion

This manual is not allways uptodate 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