Aoraï Plugin Tutorial

(A.k.a. LTL to ACSL)

Nicolas Stouls
Nicolas.Stouls@insa-lyon.fr

August 28, 2009

Foreword

The Aoraï plugin provides a method to automatically annotate a C program according to an automata 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.

Notes:

- 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.
- Aoraï has an optional dependency to ltl2ba tool, but you don't need it if you won't use the ltl syntax for automata's description.

Contents

Chapter 1

Introduction

1.1 Quick 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)
```

If you want use ltl syntax (it's not an obligation), you need to install[1] the ltl2ba tool in your current path. To enable the new syntax, 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
```

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

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

1.2 Interest of Aoraï

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

- the high level of abstraction help to write simple automate and avoid the necessity to compute all possibility of a function[2]
- thanks to the collaboration between human and plugin principle, you can easily check complex C program (see section 4.2)

1.3 Documentation's description

This document is divided into four part (each part is a paragraph from two to five):
First part is a quick overview of Aoraï . After it, you must be able to launch a basic verification and know the general principle of the software
The second paragraph is for basic user with a description of Aoraï languages which describes automates to create your own specification for verification.
The third explains how to prove a program with a user and human collaboration
Finally, the last paragraph explains Aoraï 's theory,software's architecture and and algorithm in order to help people who wants to contribute.

[2]for more information, see chapter 5

Chapter 2

Quick overview

In this chapter we will see how to use frama-c and the couple Jessi-Aoraï to prove that a C program has the same comportment to an automate.

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 automate, which will be described in the second part. In fact,if we consider we have already write the file which described the automata.

Jessie's verification[2] can only be done on C annotated code[3], that's why Aoraï create a new C file with the automata which is integrated to the source.

If you look in the example's archive, you will find three files:

- example.ltl and example.ya which are equivalent and give a description of the automata's specifications.
- example.c is the implementation which will be checked.

With two files (automata's description and C file), we can create an annotated file[4] in order to process the validation with Jessie plugin.
The command is `frama-c example.c -ltl-automata example.ya`.

In order to decide if the original program is correct fro the automata, it is sufficient to establish than the generated C is valid. For instance, with the Jessie plugin:
`frama-c example_annot.c -jessie`.

[1] From `http://frama-c.cea.fr/ltl_to_acsl.html`

[2] For more information about Jessie and code verification,please refer to `http://frama-c.cea.fr/jessie.html`

[3] Syntax are explained in section 4.1

[4] more information about annotation are available in section 4.1

2.1.2 Automata and verification

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

For example, in the example, the automata is described by figure 2.1.

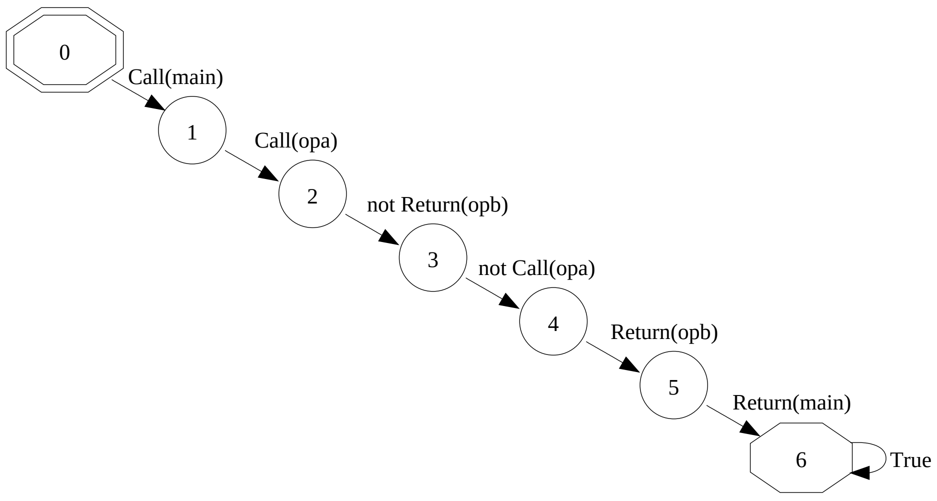

Figure 2.1: Automata

With description from files like ya or ltl, a specification is computed for each operation, in terms of states and transitions from the automata. For instance, the following specification correspond to the previous automata:

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

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

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

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

And 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)* (eg. in our point of view, the pre or the post condition of a C function are defined by the set of states authorized just before/after the call, as such as the set of crossable transitions).

Finally, Aoraï generates a new C program, including the automata axioma-

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

/*@ requires rr==1;
  @ behavior j :
  @ ensures rr==2;
*/
void opa() {rr++;}

/*@ requires rr>=1 && rr<=500;
  @ behavior f :
  @ ensures rr<600;
*/
void opb () {rr+=2;}

/*
  @ behavior g :
  @ ensures rr==600;
*/
void opc () {rr=600;}
```

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

Figure 2.2: Example of C File

tization, 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 properties.

Some times, automata has not enough information to check the validity of the C-program (as we seen in the previous chapter), and the problem is only related to the implementation which is used. In this case you can add some properties in the automata *or* in the generated files. For more information about that, please read the 4.2 section.

2.2 Help Command

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-automata <f>	Considers the property described by the ya automata (in Ya language) from file <f>
-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 states and transitions.
-ltl-dot	Generates a dot file of the automata. Dot is a graph format used by the GraphViz tool[5].

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

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

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.
- In the init state from the automate, condition on C-array or C-structure are not statically evaluated (it's an optimization) but are supported.

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

Chapter 3

Aoraï 's Languages

All the Aoraï 's versification's principle is built from the automata, that's why the plugin has languages to write automata. The easiest syntax is probably the YA which was created for Aoraï but for compatibility reason some others are supported like LTL or PROMELA.

3.1 YA

The description of the automate can be performed by many ways, but we recommend to use the one which is explained below:

- Initial states of automaton are specified using the *%init* keyword followed by a coma separated list containing the state name

```
%init:  S1, S2, ..., Sn;
```

- Acceptance states are specified using the *%accept* keyword followed by a coma separated list containing the state name

```
%accept:  S1, S2, ..., Sn;
```

- States and transitions are described by sets of the following form

```
state :    condition_1  -> new_state_1
|  condition_2  -> new_state_2
|  condition_n  -> new_state_n
  ;
```

  if a condition is always true, it can be omitted with its surrounding parenthesis:

```
state:  -> new_state
  ;
```

- Condition is a logical expression based on the C syntax:
  - identifier are global variable from the verified program
  - CALL, RETURN and COR are functions taking as parameter the function name of the verified program, and testing respectively the call, the return, the call or the return of this function
- For example, for the automata which is used in the chapter 2.1

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

The call is done through `frama-c prog.c -ltl-automata formula.ya`.

3.2 LTL

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 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.

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.Furthermore, the LTL formula for the example in chapter 2.1 is write in figure 3.4

/ Formula */*
F ::=

(1st 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

Atomicity Property

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

Figure 3.3: Concrete example of LTL formula

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

Figure 3.4: LTL formula for chapter 2.1

3.3 PROMELA

TODO

Chapter 4

Advanced Features

4.1 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[1]) completed with 6 types of information:

- An axiomatization of automata associated to the property (Sect. 4.1.1);
- Some variables modellizing the current states and transitions of the automata (Sect. 4.1.2);
- Some invariants characterizing links between program specification and automata (Sect. 4.1.3);
- Additional pre and post-conditions for each operation, in terms of the states and transitions of the automata (Sect. 4.1.4);
- Some piece of ghost code before each call and each return statement, which updates the current state of the automata (Sect. 4.1.5);
- Loop invariants in terms of the automata (Sect. 4.1.6).

For each if these information we give (figure 4.1 to 4.5) a piece of the C file generated according to the example from section 2.1.

4.1.1 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 :

[1]ACSL language for annotation is described at `http://frama-c.cea.fr/acsl.html`

- 2 logic functions that associate, to a transition number, its starting or ending 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 4.1.

```
/*@ axiomatic transStart {
  @   logic integer transStart(integer tr) ;
  @   axiomtransStart0: (transStart(0) == 0);
  @   axiomtransStart1: (transStart(1) == 1);
  @   ...}
*/
/*@ axiomatic transStop {
  @   logic integer transStop(integer tr) ;
  @   axiomtransStop0: (transStop(0) == 1);
  @   axiomtransStop1: (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 4.1: Example of Automata Axiomatization

4.1.2 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 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 ending states of these transitions. An example is given figure 4.2.

4.1.3 Invariants

Some invariants are used to join model variables and to link the specifications of the automata and of the program. For instance, the invariant given figure 4.3

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 4.2: Example of Generated Variables

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.

/*@ ***global invariant*** Unreachability$_1$:
@ $\forall st; 0 \leq st < NbStates \ \wedge$
@ $\left(\begin{array}{l} \forall tr; 0 \leq tr < NbTrans \\ \qquad \Rightarrow curTr[tr] = 0 \ \vee \ transStop(tr) \neq st \ \vee \\ \qquad\quad \neg transCond(tr) \ \vee \ buch\_CurStates\_old[transStart(tr)] = 0 \end{array}\right)$
@ $\Rightarrow curSt[st] = 0$;
*/

Figure 4.3: Example of Generated Invariant

4.1.4 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.

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 4.4.

4.1.5 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 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;

requires $0 == \mathit{curTr}[0] \land 0 == \mathit{curTr}[2] \land 0 == \mathit{curTr}[3] \land 0 == \mathit{curTr}[4] \land 0 == \mathit{curTr}[5] \land 0 == \mathit{curTr}[6]$
requires $0 \mathrel{!=} \mathit{curTr}[1]$
requires $0 == \mathit{curSt}[0] \land 0 == \mathit{curSt}[1] \land 0 == \mathit{curSt}[3] \land 0 == \mathit{curSt}[4] \land 0 == \mathit{curSt}[5] \land 0 == \mathit{curSt}[6]$
requires $0 \mathrel{!=} \mathit{curSt}[2]$
behavior buch_0:
  assumes $0 \mathrel{!=} \mathit{curSt}[2]$
  ensures $0 == \mathit{curTr}[0] \land 0 == \mathit{curTr}[1] \land 0 == \mathit{curTr}[3] \land 0 == \mathit{curTr}[4] \land 0 == \mathit{curTr}[5] \land 0 == \mathit{curTr}[6]$
  ensures $0 \mathrel{!=} \mathit{curTr}[2]$
  ensures $0 == \mathit{curSt}[0] \land 0 == \mathit{curSt}[1] \land 0 == \mathit{curSt}[2] \land 0 == \mathit{curSt}[4] \land 0 == \mathit{curSt}[5] \land 0 == \mathit{curSt}[6]$
  ensures $0 \mathrel{!=} \mathit{curSt}[3]$

Figure 4.4: Example of Generated Specifications for *opa*

- 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 cross conditions. Figure 4.5 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 4.5: Example of Generated Synchronization Code

4.1.6 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 4.6

```
/*@ 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 4.6: Example of Generated Loop Invariants

4.2 Interaction with Annotated Files

TODO

Chapter 5

Going Further

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.

5.1 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 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 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.

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

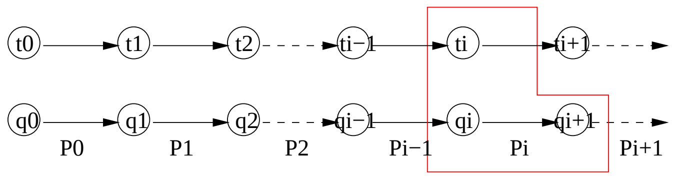

Figure 5.1: Synchronization of Paths from automata and from Program

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 1 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 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 has to update the set of states synchronized with the current state.

Synchronization The synchronization condition can be expressed with an invariant which verify 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 the inclusion of the reachable states into the acceptance states set.

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

5.2.1 Automata Modellization

In order to link models from the program and the property, we describe the 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-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 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.

5.3 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 automata, and next we propagates the generated constraints in order to converge into a fix-point of specifications.

5.3.1 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 always 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.

5.3.2 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 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.

5.4 Plugin Architecture

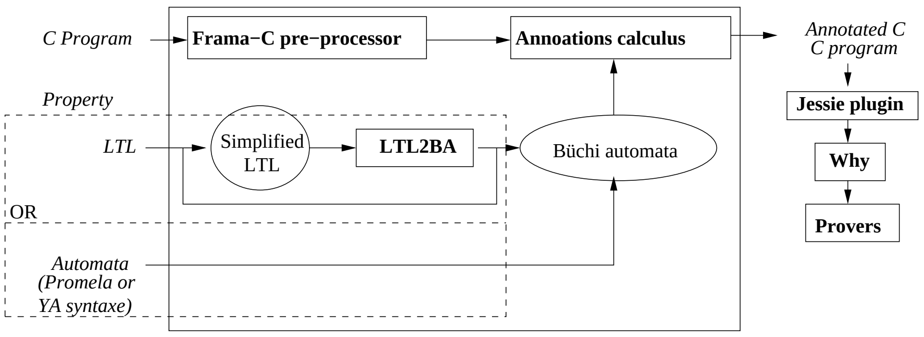

Figure 5.2: Plug-in Structure

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).

Chapter 6

Conclusion

This manual is not always 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