A Programmer's Guide to AspectLTL
(Plug-in version 1.1.3, December 2010)

Shahar Maoz[1] and Yaniv Sa'ar[2]

[1] RWTH Aachen University, Germany
maoz@se-rwth.de
[2] The Weizmann Institute of Science, Israel
yaniv.saar@weizmann.ac.il

Abstract. This document is a programmer's guide to AspectLTL [2], a temporal-logic based language for the specification and implementation of crosscutting concerns. The guide provides an overview of AspectLTL from a programmer's perspective, covering the main constructs and features of the language using a number of examples, together with an overview of the main features of the AspectLTL plug-in.

1 Introduction

1.1 What is AspectLTL?

AspectLTL [2] is a language for the specification and implementation of crosscutting concerns, based on linear temporal logic (LTL) [4]. The aspects of AspectLTL, called LTL aspects, enable the declarative specification of expressive crosscutting concerns. These include the specification of safety properties, which may be used to prevent a base system from visiting 'bad states', the specification of liveness (fairness) properties, which may be used to force a base system to visit 'good states' (infinitely often), and the addition of new behaviors to a base system, which is done by specifying the existence of new transitions and new states as necessary.

AspectLTL comes with a synthesis-based weaving process, whose output is a correct-by-construction executable artifact. Following a composition of the specified aspects with a base system, using symbolic disjunctive and conjunctive operations, we formulate the problem of correct weaving as a synthesis problem [5], essentially a game between the environment and the (augmented) base system. An algorithm based on [3] is used to solve the game, that is, to provide the augmented system with a winning strategy, if any.

If a winning strategy is found, it is presented as a deterministic, executable automaton, which represents an augmented base system whose behavior is guaranteed to adhere to the specified aspects, in all possible environments. If a winning strategy is not found, we know that it does not exist, that is, that no system exists which is based on the base system and can adhere to the specified LTL aspects in all environments. Thus, LTL aspect composition and synthesis is sound and complete.

An AspectLTL specification is made of a base system, given as a finite-state machine specified in an SMV [1] format, and a set of LTL aspects, each of which is specified in a similar SMV-like format, containing a symbolic representation of the aspect's added behaviors (transitions) and a related LTL specification.

AspectLTL is supported by an Eclipse plug-in, developed on top of JTLV [6], a framework for the development of verification algorithms, using BDD-based symbolic mechanisms. The update site for the AspectLTL plug-in is http://aspectltl.ysaar.net/updatesite/. The AspectLTL website is http://aspectltl.ysaar.net/.

1.2 What is this guide?

This document is a programmer's guide to AspectLTL, covering the constructs and features of the language from a programmers perspective, using a number of examples. To follow the examples it is recommended to download and install the AspectLTL plug-in[1]. The examples are available with the plug-in.

The guide is not aimed to provide a full and formal account of the language and its semantics. For the interested reader, a formal account is available in [2].

1.3 Prerequisites: what should you know before you start?

As AspectLTL is an aspect language for LTL specifications, this guide assumes that the reader have some familiarity with LTL, and aspect oriented terminology. In addition, since AspectLTL uses a variant of SMV, the language used by the SMV model-checker, some familiarity with the SMV language is recommended. However, knowledge of LTL synthesis algorithms, of BDD technology, and of the SMV model-checker, is not required.

1.4 What's the structure of this guide?

The guide starts with a simple HelloWorld example. Section 3 describes additional example AspectLTL programs. The main features of the AspectLTL plug-in are described in section 4. The appendices contain the language's list of keywords, list of logical operators, and list of temporal operators.

2 Your first AspectLTL program: `HelloWorld`

An AspectLTL program is made of a base and a set of aspects. The base specifies a state machine. Each aspect may specify transitions that can be added to the base state machine and LTL formulas that the weaved program should satisfy.

We will start out by looking at an example program that we'll build out of a base and two aspects. This example will give us something concrete and simple to talk about (it is not meant to motivate the use of AspectLTL and demonstrate its power).

Listing 1.1 shows the base of our first example program: `HelloWorld`.

[1] AspectLTL plug-in was tested on Eclipse Helios 3.6.1 and on Eclipse Galileo 3.5.1. The plug-in should be compatible with earlier versions.

```
1  -- base for HelloWorld example
2  MODULE HelloWorldBase
3    VARENV
4      exe : boolean;
5    VAR
6      state : {start, prologue, printHello, epilogue, exit};
7    INIT
8      (state=start);
9    TRANS
10     next(state) = case
11       state = start & exe      : prologue;
12       state = prologue         : printHello;
13       state = printHello       : epilogue;
14       state = epilogue         : exit;
15       1                        : state;
16     esac;
```

Listing 1.1. The `HelloWorldBase` base

Every AspectLTL program must have exactly one base. An AspectLTL base is saved in a file with a `.bltl` extension. In our case, the base is called `HelloWorldBase` and it is saved in a file named `HelloWorldBase.bltl`. The base starts with the keyword `MODULE` followed by its name.

The base defines environment and system variables and a related transition system. Environment variables are input variables: their value is set by the environment. System variables values are set by the system. The keyword `VARENV` is used to define a section for environment variable definitions. The keyword `VAR` is used to define a section for system variable definitions. In `HelloWorldBase` we define a single Boolean environment variable `exe` and a single system variable `state` whose domain includes five values.

Next, in the `INIT` section, the base defines the initial state of its transition system: the initial value of the system variable `state` is `start`. Note that this is not an assignment but a logical proposition.

Finally, the transitions of the system are defined in the section `TRANS`. In our example, this is done by relating the next value of the `state` variable: if `state` equals `start` and the Boolean variable `exe` is true, the next value of `state` will be `prologue`; if `state` equals `prologue`, the next value of `state` will be `printHello`; if `state` equals `printHello`, the next value of `state` will be `epilogue`; if `state` equals `epilogue`, the next value of `state` will be `exit`; and in all other cases, state will not change its value.

Alternatively, one can use an `ASSIGN` section to define the transitions. This is demonstrated in other examples. The `ASSIGN` section relates a variable to its the next step assignment, whereas the `TRANS` and `INIT` sections are propositional style declarations. Choosing between `ASSIGN` and `TRANS` is a matter of convenience and programming style; it does not make a difference in terms of the language's expressiveness.

We are now ready to write our first AspectLTL aspect. On top of the base, we would like to specify an aspect that adds a state before printing and notifies us when it reaches this state. We call it `HelloWorldBeforePrintAspect`.

```
1  -- adding a state before print
2  ASPECT HelloWorldBeforePrintAspect
3    VAR
4      ext state : {prologue, printHello};
5      new beforePrint : boolean;
6    TRANS
7      state = prologue & next(state) = prologue &
8          !beforePrint & next(beforePrint);
9    TRANS
10     state = prologue & next(state) = printHello &
11         beforePrint & !next(beforePrint);
12   LTLSPEC
13     -- initialize beforePrint to false
14     !beforePrint &
15     -- if in prologue and not beforePrint, stay in prologue
16     [] (((state = prologue) & !beforePrint)
17         -> (next(state) = prologue))  &
18     -- if not in prologue then beforePrint stays idle
19     [] ((state != prologue)
20         -> (beforePrint = next(beforePrint)));
```

Listing 1.2. The `HelloWorldBeforePrintAspect` aspect

The aspect `HelloWorldBeforePrintAspect` begins with the keyword `ASPECT` followed by its name. It defines the system variables it uses in the `VAR` section. The keyword `ext` is used to reference external variables, that is, ones that are defined in the base or in other aspects. The keyword `new` is used to define new variables. Similarly, the environment input variables can be defined in `VARENV` section.

We use the `TRANS` section to define new transitions. In this aspect, there are two transitions from the state `prologue`: when `beforePrint = 0`, the system can stay in `prologue` and set `beforePrint` to 1; when `beforePrint = 1`, the system can move to state `printHello` and set `beforePrint` to 0.

We use the `LTLSPEC` section to define LTL specifications that the weaved controller must satisfy. In this aspect we initialize the `beforePrint` to 0, we define a safety formulas that requires that when we are in state `prologue` and `beforePrint = 0`, the system will stay in `prologue` in the next state, and another safety formula, to require that when we are not in state `prologue`, the variable `beforePrint` will not change in the next state. Note that otherwise (i.e. in state `prologue` and `beforePrint = 1`), the transition is not restricted, namely the transition defined at the base will be enabled.

```
1  -- requires to reach the exit
2  ASPECT HelloWorldReachExitAspect
3    VAR
4      ext state : {prologue, exit};
5
6    LTLSPEC
7      -- whenever in prologue eventually exit
8      [] ( (state=prologue) -> <> (state=exit));
9      -- whenever in start eventually exit (unrealizable
10     -- because exe is controlled by the environment)
11     -- [] ( (state=start) -> <> (state=exit));
```

Listing 1.3. The `HelloWorldReachExitAspect` aspect

Another aspect, named `HelloWorldReachExitAspect` is shown in Listing 1.3. This aspect adds no variables or transitions. It only specifies a single *response* formula: whenever in state `prologue`, eventually the state must be `exit`.

This aspect may be required if there is some self-transition on one of the states, which allows the system to stay in that state forever. Also note the other response formula, which is commented out. This formula requires that whenever in state `start`, eventually the state must be `exit`. This however cannot be guaranteed by the system because the environment controls the variable `exe`: if it is not set, the system will stay in state `start` forever. Therefore, the commented formula makes the specification unrealizable.

3 Additional Examples

TBD

4 The AspectLTL Plug-In

AspectLTL is supported by an Eclipse plug-in. The update site for the plug-in is http://aspectltl.ysaar.net/updatesite/.

Below we list and explain the various features of the AspectLTL plug-in.

4.1 Built-in examples

AspectLTL plug-in comes with wizards to create new base or aspect files. In addition, the plug-in provides a set of built-in example specifications. To open an example, click `new/examples` and choose one of the AspectLTL examples. A wizard dialog will open, where you choose the folder in which the example's files will be created. When done, the example's files (typically a base and a set of aspects) are copied into the folder you chose and are automatically opened in two separate editor panes.

Version 1.1.3 (December 2010) includes the following examples:

Counter Counting with aspects.
Exam service Students exam service example from [2].
Farmer The classic river crossing puzzle.
Hello world Simple hello world (described in the programmer's guide).
Synchronized mutex mutual exclusion of synchronous processes
Traffic Cars moving in two lanes with obstacles.
Traffic (modular) A modular variant of the base of the `Traffic` example.

4.2 What can you do with an AspectLTL specification?

An AspectLTL specification is made of a base file (extension `.bltl`) and a number of aspects (extension `.altl`). When selecting a base and at least one aspect file, the AspectLTL action item in Eclipse's popup menu will show the following options: `Check Realizability`, `Set as Working Set for Automatic Realizability`, `Synthesize Textual Automaton`, and `Synthesize and Generate Java Code`.

Checking realizability Checking realizability means to check whether the selected specification is realizable, i.e., is there an implementation that realizes it. Checking realizability is relatively fast. Its output is shown on the Console tab where it says whether the specification is realizable and adds some running time statistics (which can be ignored).

We recommend checking realizability often. If your specification is not realizabile, you know it should be corrected.

Set as Working Set for Automatic Realizability To support our recommendation, we allow to mark the selected specification as a "working set". Upon every save action to one of the marked files, a realizability check will be executed.

Synthesis of textual automaton This action starts with checking realizability. If the specification is unrealizable it stops and shows a notification. If the specification is realizable, a concrete implementation is produced and displayed as an automaton in textual format.

The textual format describes the states of the automaton and their possible successors. The states are numbered. The values of all variables (environment and system) in each state are shown. Note that the automaton is deterministic with respect to the environment variables: multiple successors at a state are the result of different values to the input variables, as can be set by the environment.

Synthesis and Java code generation This action starts with checking realizability too. If the specification is unrealizable it stops. If the specification is realizable, a concrete implementation is produced and a corresponding Java implementation is generated. The generated Java file is a very simple standalone Java application with a `main` method.

When executed, the program follows the behavior of the synthesized controller in 'step mode'. At every step, it shows the current state (assignments to current input and system variables), asks for the user's input on the next values to give to environment variables, and then moves to the next state.

Thus, the generated Java program allows you to play the role of the 'environment' while the computer plays the 'system'. This way you can interactively check that the generated program indeed implements the specification.

5 Appendices

5.1 AspectLTL keywords

`ASPECT` Define an aspect
`TRANS` Define a transition relation (in the base or in an aspect)
`LTLSPEC` Define an LTL specification in an aspect
`ext` Define variable as external in an aspect
`new` Define variable as new in an aspect
`MODULE` Define a module in the base
`INIT` Define initial state by assigning values to system variables (in the base or in an aspect)
`VARENV` Define environment controlled (input) variables (in the base or in an aspect)
`ASSIGN` Define an initial state or a transition relation (in the base), in an assignment programming style
`DEFINE` Define syntactic macros (in the base or in an aspect)

We list the logical operators and temporal operators supported by AspectLTL.

Logical operators

```
NOT             : '!'
AND             : '&'
OR              : '|'
IMP             : '->'
IFF             : '<->'
XOR             : 'xor'
```

Logical constants

```
TRUE            : 'TRUE'
FALSE           : 'FALSE'
```

Arithmetic operators

```
PLUS            : '+';
MINUS           : '-';
TIMES           : '*';
DIVIDE          : '/';
MOD             : 'mod';
EQUAL           : '=';
NOTEQUAL        : '!=';
LE              : '<=';
GE              : '>=';
LT              : '<';
GT              : '>';
```

Temporal operators

```
FINALLY         : '<>' | 'F' | 'FINALLY' | 'EVENTUALLY';
ONCE            : '<_>' | 'O' | 'ONCE';
GLOBALLY        : '[]' | 'G' | 'GLOBALLY' | 'ALWAYS';
HISTORICALLY    : '[_]' | 'H' | 'HISTORICALLY';
NEXT            : 'next';
PREV            : '(_)' | 'Y' | 'PREV';
UNTIL           : 'Until' | 'U' | 'UNTIL';
SINCE           : 'Since' | 'S' | 'SINCE';
RELEASES        : 'Awaits' | 'V' | 'RELEASES';
TRIGGERED       : 'Backto' | 'T' | 'TRIGGERED';
```

Misc.

```
CASE            : 'case';
END_CASE        : 'esac';
COMMENT         : '//' | '--';
```

5.2 LTL subset supported by AspectLTL

AspectLTL `LTLSPEC` is a conjunction of formulas made of the constructs defined below.

`assertion` Any boolean combination of the variables and the arithmetic operators

`past` Any formula involving assertions and past operators

`safety` Any formula of the form $\Box(\varphi)$ where φ is an assertion or past formula

`justice` Any formula of the form $\Box \Diamond(\varphi)$ where φ is an assertion or past formula

`response` Any formula of the form $\Box(\varphi \rightarrow (\Diamond \psi))$ where φ and ψ are assertions or past formulas

References

1. SMV model checker.
   http://www.cs.cmu.edu/~modelcheck/smv.html.
2. S. Maoz and Y. Sa'ar. AspectLTL: An aspect langauge for LTL specifications. In S. Chiba, editor, *AOSD*. ACM, 2011. To appear.
3. N. Piterman, A. Pnueli, and Y. Sa'ar. Synthesis of Reactive(1) Designs. In *VMCAI*, volume 3855 of *LNCS*, pages 364–380. Springer, 2006.
4. A. Pnueli. The temporal logic of programs. In *FOCS*, pages 46–57. IEEE, 1977.
5. A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, pages 179–190, 1989.
6. A. Pnueli, Y. Sa'ar, and L. Zuck. JTLV: A framework for developing verification algorithms. In *CAV*, 2010.