MAG-1.0 User Manual

INRIA, 35042 Rennes Cedex, France

Tool Name : MAG - Monitor and Aspect Generator

Owened by : INRIA, France

Realeased in : October 2014.

Description : MAG is a tool for generating monitored versions of SystemC models in order to perform statistical model checking with Plasma-Lab. MAG is based on the techniques in the CHIMP tool by Sonali Dutta, Deian Tabakov, and Moshe Y. Vardi.

Download : https://project.inria.fr/plasma-lab/mag_manual/

Platform : MAG can be installed and run in any unix environment (e.g., Ubuntu Linux, Mac OS,...).

License : MAG uses GNU GPL Version 3.

1 Components

MAG consists of 3 following components:

- MAG-1.0: This component is reponsible to automatically generate the monitors and the aspect advice file for instrumentation in order to observe execution traces of the SystemC model.
- AspectC++-1.2: It is used by MAG to instrument the model-under-verification (MUV).
- SystemC-2.3.0_modified: The patched version of OSCI kernel-2.3.0. The patach enables user-defined temproal resolutions (sampling rates) during observing execution traces.

2 Installing MAG

To install and run MAG, the following packages are required:

- boost library
- autoconf 2.61 and later
- automake 1.10 and later
- libtool 2.4 and later

- GNU make 3.81 and later
- Java SDK or JRE 1.6 and later
- g++ with build-essential package
- libpuma-dev package used by AspectC++
- GNU Scientific Library 1.6 and later for running the example code

To install run *make* (to compile MAG) and run *make clean* to uninstall MAG. In order to make the instrumented SystemC model run, the modified version of SystemC is needed. To install SystemC, please refer to the INSTALL file in the provided SystemC package.

3 Running MAG

To run MAG, users need to write a configuration file first containing all properties to be verified with the declarations of all used primitives, as well as other necessary information. From this configuration, MAG generates the monitors and an aspect-advice file that is then used by AspectC++ to generate the instrumented model. Finally, the instrumented MUV code and the monitor are compiled together and linked with the SystemC kernel into an executable model.

The executable model resulting from the first step is run through our plugin to execute the simulation with the inputs provided by users. For every property, Plasma-lab checks the validity of the property according to the execution traces produced by the corresponding monitor.

3.1 Generating Monitor with MAG

User runs MAG to generate the monitor and aspect advices according to the property to be verified.

- Write the configuration file according to the verifying property (see Section 4 for more details)
- Change to the directory of the MAG tool and run

$$mag \quad -conf \quad \text{path to configuration file}$$

- MAG will generate three files: header and source files of the monitor, and the aspect advice file "aspect_definitions.ah" in the user-defined location

3.2 Instrumenting with AspectC++

User runs AspectC++ with the MUV, the generated monitor, and the aspect advice file to instrument the MUV.

- Change the working directory to AspectC++-1.2
- Run the following command to generate puma.config file in the working directory

$$ag++ \quad --gen\_config$$

– Copy the header and source of the generated monitor and the aspect advice file "aspect_definitions.ah" inside the source directory of the MUV
– For each header or source file of the MUV, user runs the following command to generate the instrumented version.

$$\begin{aligned} ac++ \; & -c\, SOURCE\_HOME/file\_name \\ & -o\, TARGET\_HOME/file\_name \\ & -p\, SOURCE\_HOME/ -I\, SOURCE\_HOME \\ & -I\, SYSTEMC\_HOME/include \\ & --config\, ASPECTC\_HOME/puma.config \end{aligned}$$

where SOURCE_HOME is the path to the source directory of the MUV, TARGET_HOME is the path to the directory where user wants the AspectC++ puts the instrumented version, SYSTEMC_HOME is the path to the patched version of systemc-2.3.0, and ASPECTC_HOME is the path to the AspectC++. Alternatively, user can use the sheel script included in the examples to make the steps above automatically.

3.3 Compiling Instrumented Code

– In the main header file of the MUV, user includes the monitor header file, for example

$$\#include \;\; "monitor\_multi\_lift.h"$$

– In the source file of the MUV, user adds the following line just before the call to *sc_start()*

$$mon\_observer* \;\; obs \;=\; local\_observer :: createInstance(1, parameters);$$

The *parameters* depends on the generated monitor, for example, in the included example of multi-lift system in the MAG tool package, it is:

$$mon\_observer* \;\; obs \;=\; local\_observer :: createInstance(1, \&liftsystem);$$

– User compiles the instrumented MUV with g++ compiler and links with the patched SystemC libary, provided in MAG package

We also provide the shell scripts in the example directory of MAG tool that automatically generated the instrumented MUV. User can modify them according to his requirements.

4 Configuration File Triggers

The configuration file is used to guide MAG generating the appropriate monitors and aspect advice file using by AspectC++. Its triggers are given as follows:

– **verbosity**: The integer value between 0 and 9. It represents the level of information messages outputing by MAG. The default value is 1. For example to define the value of verbosity as 6. User can write:

$$verbosity\ \ 6$$

– **output_file**: The path to the source file of the generating monitor. By default, MAG will generate a file *monitor.cc* in the working directory of MAG. For example:

$$output\_file\ \ /home/user/model/my\_monitor.cc$$

– **output_header_file**: The path to the header file of the generating monitor. The default header file is based on the name of the output_file without extension. For example:

$$output\_header\_file\ \ /home/user/model/my\_monitor.h$$

– **mon_name**: The name of the generated monitor. The default is monitor. For example:

$$mon\_name\ \ my\_monitor$$

– **plasma_file**: The path to the generated Plasma Lab project file. For example:

$$plasma\_file\ \ /home/user/PLASMA\_Lab-1.3.0/my\_project.plasma$$

– **plasma_project_name**: The generated Plasma Lab project name. For example:

$$plasma\_project\_name\ \ my\_project$$

– **plasma_model_name**: The Plasma Lab model. For example:

$$plasma\_model\_name\ \ my\_model$$

– **plasma_model_content**: The path to the executable SystemC model. For example:

$$plasma\_model\_content\ \ /home/user/model/instrumented/muv$$

– **write_to_file**: Write execution traces to a file. For example, user does not want to log the traces to file:

$$write\_to\_file\ \ false$$

– **include**: If the verifying property uses references to an object of a class or a module. Then this trigger indicates the header file that needs to be included in the header file of the monitor. For example, there is a reference to the module A that is declared in the header file A.h, then we define the include trigger as follows:

$$include\ \ A.h$$

– **usertype**: Consider an object of type class A, user wants to refer to attributes of this object. These attributes can be protected, private, public, or accessed by some getting methods. To make the monitor generated by MAG can access these attributes, user uses the usertype trigger as follows:

$$usertype \;\; A$$

– **type**: If the verifying property uses references to an object of class or module of type T, users need to tell MAG that this object has type T. To do that user defines the value of type trigger. For example, consider a property that refers to two object pointers of types class A and B, respectively. Then user defines the trigger as follows:

$$type \;\; A * a$$
$$type \;\; B * b$$

– **attribute**: The value of this trigger defines which attributes of an object that user wants to observe the values during the execution of the model. The trigger syntax is *attribute attribute_name label*. For example, assume that user wants to observe the value of the private attribute t of the above object pointer a of type class A and labels it with a_t. User can write:

$$attribute \;\; a \rightarrow t \; a\_t$$

– **att_type**: For each defined attribute, user needs to define its type. MAG supports all primitive datatypes of SystemC and C++ except char and *string*. The trigger syntax is *att_type type_name attribute_label*. For example, consider the above attribute a_t, assume that it is of type *int*. User write the following trigger in the configuration file:

$$att\_type \;\; int \; a\_t$$

– **eventclock**: The value of this trigger defines a Boolean variable that is true immediately when a specific *event* is notified. This variable ussually is used to define a temporal resolution. For example, consider an event e of the object a. The following Boolean variable $e\_notified$ is set to be true whenever e is notified.

$$eventclock \;\; a \rightarrow e.notified \; e\_notified$$

– **location**: The value of this trigger defines a Boolean variable that is true whenever a location in the source code model will arrive during the simulation. Location trigger provides four primitives *entry*, *exit*, *call*, and *return* that refer to the location immediately before the first executable statement, the location immediately after the last executable statement in a function, the location that contains the function call, and the location immediately after the function call, respectively. The syntaxes of these primitives in the configuration file are given as follows:

$$location \; location\_variable\_name \; function\_pattern : entry$$
$$location \; location\_variable\_name \; function\_pattern : exit$$
$$location \; location\_variable\_name \; function\_pattern : call$$
$$location \; location\_variable\_name \; function\_pattern : return$$

where the *function_pattern* follows the same as pointcut expressions in AspectC++. The *function_pattern* has the form $return\_type\ \ class\_name :: funtion\_name(argument\_list)$.

- **plocation**: This trigger helps user define a Boolean variable that holds the value true immediately before or after the execution of all statements that match the value of the trigger (defined as a regular expression). The syntax of this trigger is $plocation\ location\_variable\_name\ statement : before$ or $plocation\ location\_variable\_name\ statement : after$. For example, we want to declare the Boolean variable $loc1$ that holds the value true immediately before the execution of all statements that contain the devision operator "/" followed by zero of more spaces, and the variable "a". We write the following trigger in the configuration file.

$$plocation\ \ loc1\ \text{“}/*a\text{”} : before$$

- **value**: Using this trigger user can define a variabble that will contain the return value, a parameter value passing to a function defined in the SystemC model. Then user can use this variable to define his formula. The syntax is given as follows:

$$value\ \ type\_of\_variable\ name\_of\_variable\ function\_pattern : i$$

It defines a variable that contains the value passed as i^{th} (i can be from 1 to the number of arguments the function has) parameter to the function.

$$value\ \ type\_of\_variable\ name\_of\_variable\ function\_pattern : 0$$

It defines a variable that contains the return value of the function.

- **formula**: The value of this trigger is a specification that contains two elements in the form $\{BLTLformula\}@\{name\}$. The firt element is a BLTL formula and the second one is the name of the specification. For example:

$$formula\ \ G <= \#100(a\_t >= 1)@property\_1$$

- **time_resolution**: The user-defined temporal resolution. For example:

$$time\_resolution\ \ MON\_TIMED\_NOTIFY\_PHASE\_END$$

- **comment**: To make a line as a comment, user puts # at the beginning of the line, then MAG will ignore it.

We also provide a dummy mm_config.txt file in the source code directory of MAG. User can modify it according to his requirements.

5 BLTL and Clock Expressions

User specifies the desired properties in bounded linear temporal logic formulas and the sampling rate of states in the execution traces by clock expressions.

BLTL is defined by the following grammar, where the time bound t that represents an amount of simulation time or a number of state changes in an execution trace (in our verification framework, it has the form $<= \#number$):

$$\varphi ::= \texttt{true} | \texttt{false} | p \in AP | \varphi_1 \wedge \varphi_2 | \neg\varphi | \varphi_1 \texttt{U}_\texttt{t} \varphi_2.$$

The temporal modalities `F` (the "eventually", sometimes in the future) and `G` (the "always", from now on forever) can be derived from the "until" `U` as $\texttt{F}_\texttt{t}\varphi = \texttt{true}\texttt{U}_\texttt{t}\varphi$ and $\texttt{G}_\texttt{t}\varphi = \neg\texttt{F}_\texttt{t}\neg\varphi$, respectively. The semantics of BLTL is defined w.r.t finite sequences of states of $\mathcal{M}$. We denote the fact that ω, a finite sequence of states, satisfies the BLTL formula φ by $\omega \models \varphi$.

- $\omega^k \models \texttt{true}$ and $\omega^k \not\models \texttt{false}$
- $\omega^k \models p, p \in AP$ if and only if $p \in L(\omega(k))$
- $\omega^k \models \varphi_1 \wedge \varphi_2$ if and only if $\omega^k \models \varphi_1$ and $\omega^k \models \varphi_2$
- $\omega^k \models \neg\varphi$ if and only if $\omega^k \not\models \varphi$
- $\omega^k \models \varphi_1 \texttt{U}_\texttt{t} \varphi_2$ if and only if there exists an integer i such that $\omega^{k+i} \models \varphi_2$, $\Sigma_{j<i}(t_j - t_{j-1}) \leq t$, and for each $0 \leq j < i, \omega^{k+j} \models \varphi_1$

The set of atomic propositions AP that is formed by the primitives that are declared in the configuration file of our framework and let users define properties about the states of user-code, and SystemC kernel. We have the following:

$$\begin{array}{lll}
SystemC_expr & ::= & model_expr | kernel_expr \\
model_expr & ::= & att_expr | loc_expr | arg_expr | proc_expr \\
loc_expr & ::= & [\texttt{before}|\texttt{after}]\{code_label| \\
 & & syntax_expr\} | func_name : \\
 & & \{\texttt{entry}|\texttt{exit}|\texttt{call}|\texttt{return}\} \\
arg_expr & ::= & func_name : nonnegative_integer \\
proc_expr & ::= & proc_name.\texttt{proc_state} \\
kernel_expr & ::= & phase_expr | event_expr \\
phase_expr & ::= & \texttt{kernel_phase} \\
event_expr & ::= & event_name.\texttt{notified}
\end{array}$$

att_expr is an expression that involves evaluations of variables including module's *protected* and *private* attributes. User uses *attribute* trigger to form *model_expr*. For example, *attribute* $m \rightarrow a$ a to observe the value of a in the module M given that the triggers usertype and type are defined as *usertype* M and *type* $M * m$, respectively.

loc_expr is an expression that uses a location in the source code of the verifying model which is defined using *location* and *plocation* triggers in the configuration file. For example, assume that we want to specify the property *"always the value of the variable a in the module M is different from 0 whenever it is used as a divisor within t seconds"*. We first define the trigger *plocation* $loc1$ "$/*a$" : $before$ that declares the Boolean variable $loc1$ that holds the value true immediately before the execution of all statements that contain the devision operator "/"

followed by zero of more spaces, and the variable "a". With the attribute above, the property is expressed as follows:

$$G_t(loc1 \rightarrow (a! = 0))$$

Another example, assume that user wants to specify the property *"*`send`() *remains blocked until* `receive`() *has returned within t seconds"*. The following triggers declare Boolean variables *send_start* and *send_done* that hold the value true immediately before and after a function call of the function *send*() of the module *producer*, respectively.

location send_start "%*producer* :: *send*()" : *call*
location send_done "%*producer* :: *send*()" : *return*

Similarly, the trigger *location rcv* "%*consumer* :: *receive*()" : *return* declares a Boolean variable *rcv* that holds the value true immediately after a function call of the function *receive*() of the module *consumer*. Using these triggers, the property is expressed as follows:

$$G_t(send\_start \rightarrow (!send\_done\ \mathtt{U_t}\ rcv))$$

arg_expr is an expression that uses the return values, parameter values passing to functions defined in the SystemC model. This expression can be formed by using the trigger *value* in the configuration file.

For each process name, the primitive *proc_expr* indicates the status of this process in the simulator kernel that can be *waiting*, *runnable*, or *running*. The *kernel_expr* consists of the primitives to expose the current state of the kernel (*phase_expr*) (e.g., end of delta-cycle notification) and when a specific event is notified (*event_expr*). For instance, the following trigger declares a Boolean variable *wevent* that holds immediately when *write_event* is notified.

$$eventclock\ wevent\ write\_event.notified$$

This variable can be used to define a temporal resolution in the configuration file of our framework.

As explained earlier, users can define their own temporal resolutions. A temporal resolution is specified by using a disjunction of events, locations or kernel phases as in the BLTL formula of the form $(clk_1|clk_2|...)$. For instance, the specification

$$time\_resolution\ \ loc1$$

will check the formula over a trace of states sampled each time the location `loc1` is reached. If user provide no clock expression, MAG will sample at all 18 predefined kernel phases. Table 1 shows the list of 18 predefined kernel phases.

6 Contact

INRIA Rennes
263 avenue du Général Leclerc, 35042 Rennes, France

Phase name	Sampling location
MON_INIT_PHASE_BEGIN	Before initialization phase begins
MON_INIT_UPDATE_PHASE_BEGIN	Before initialization update phase begins
MON_INIT_UPDATE_PHASE_END	After initialization update phase ends
MON_INIT_DELTA_NOTIFY_PHASE_BEGIN	Before initialization delta notification phase begins
MON_INIT_DELTA_NOTIFY_PHASE_ENDS	After initialization delta notification phase ends
MON_INIT_PHASE_END	After initialization phase ends
MON_DELTA_CYCLE_BEGIN	Before a delta cycle begins
MON_DELTA_CYCLE_END	After a delta cycle ends
MON_EVALUATE_PHASE_BEGIN	Before an evaluation phase begins
MON_EVALUATE_PHASE_END	After an evaluation phase ends
MON_UPDATE_PHASE_BEGIN	Before an update phase begins
MON_UPDATE_PHASE_END	After an update phase ends
MON_DELTA_NOTIFY_PHASE_BEGIN	Before a delta notification phase begins
MON_DELTA_NOITIFY_PHASE_END	After a delta notification phase ends
MON_TIMED_NOTIFY_PHASE_BEGIN	Before a timed notification phase begins
MON_TIMED_NOTIFY_PHASE_END	After a timed notification phase ends
MON_METHOD_SUSPEND	After an sc_method has ended execution
MON_THREAD_SUSPEND	After an sc_thread has suspended

Table 1: Predefined Kernel Phases