4. MP Event Basics and Vocabulary

This section covers event patterns (event grammar rules) and simple behavior models: sequence (precedence) and alternatives. Recall that a sequence implies ordering. MP also supports sets of unordered events (concurrence) which will be covered later. The terms **schema, root, atomic and composite** are also explained as they relate to MP.

The MP behavior model is based on the concept of *event as an abstraction of activity.* The event has a beginning and an end, and may have duration (a time interval during which the action is accomplished). Read Section 5 of the MP Language Manual for the details of event grammar rules.

Section 5.2 of the MP Language Manual contains the notation for patterns of events. We will be covering these notations as we go in this tutorial. An excellent summary of event grammar can be found at

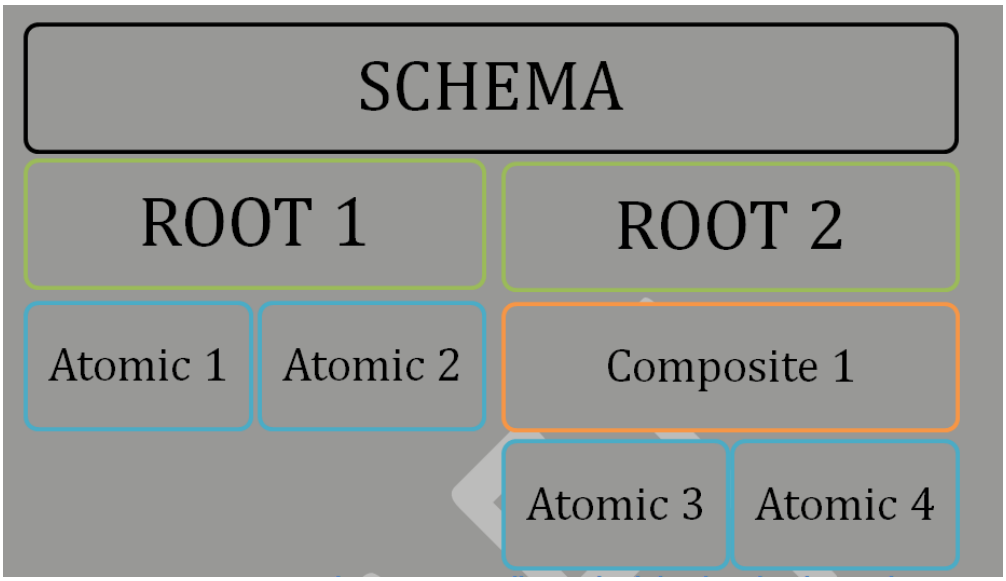
https://wiki.nps.edu/display/MP/Event+Grammar

ROOT, SCHEMA, Atomic and Composite Events

From Section 2.3 of the MP syntax manual:

"The behavior of a particular system is specified as a set of possible event traces using a *schema*. The purpose is to define the structure of event traces in terms of IN and PRECEDES relations using event grammar rules and other constraints. A schema contains a collection of events called *roots* representing the behaviors of parts of the system"

An atomic event is a single event which is not presented as decomposed (does not have any "child events") in the model. This does not mean that it could not be decomposed at some later time, just that you don't choose to do so now. A composite event is an event which consists of one or more child events (atomic or other composite events). Composition in MP is the "IN" relationship between events.



In other words:

- A SCHEMA is a description of the whole system
- A ROOT is a description of the behavior of each subsystem
- Composite and/or atomic event(s) define the behavior(s) which make up the ROOT(s)

The following example illustrates:

- ROOT
- Composite and atomic events
- "or" (alternative events) syntax
- : and ; and | syntax elements
- /* comments */

/* This a “weather example” with not much detail */

ROOT Person**: (Check_the_weather_forecast);**

Check_the_weather_forecast :

(Take_the_umbrella |

Take_the_sunglasses) Hit_the_road;

Example 1 Simple Weather (basic syntax, composite and atomic events)

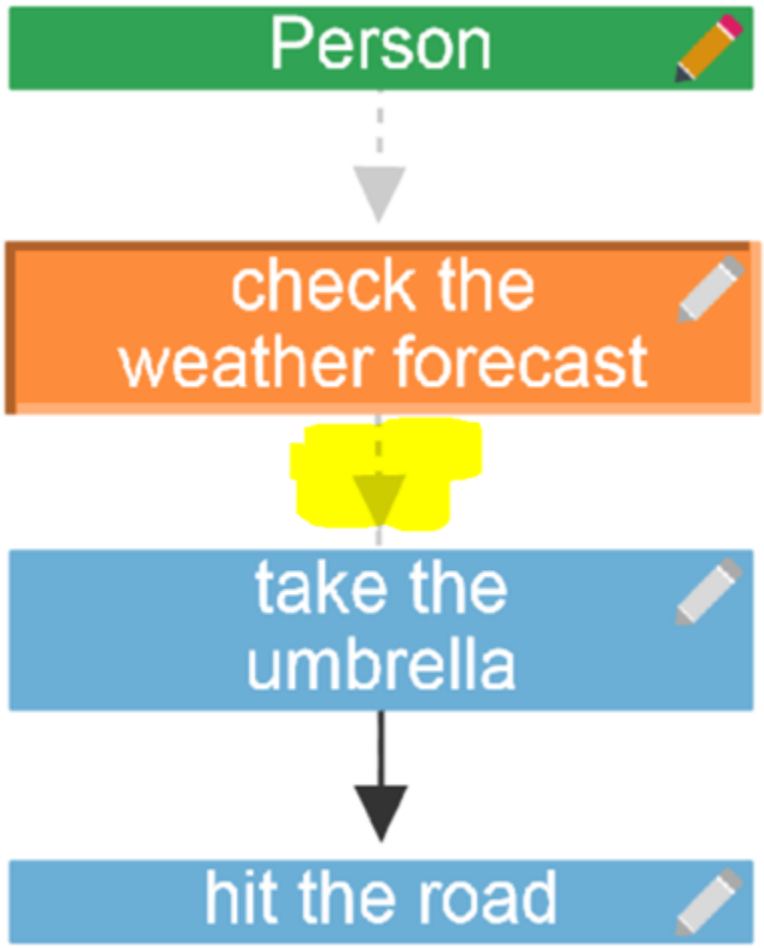
In the weather example, Take_the_umbrella, Take_the_sunglasses and Hit_the_road are atomic events. Check_the_weather_forecast is a composite event. "or" is indicated by | between the atomic events. Notice the positioning of the required : and;. These are required parts of MP grammar and failure to include them is a common source of beginning syntax errors. Also note that ROOT (Person) is a composite event.

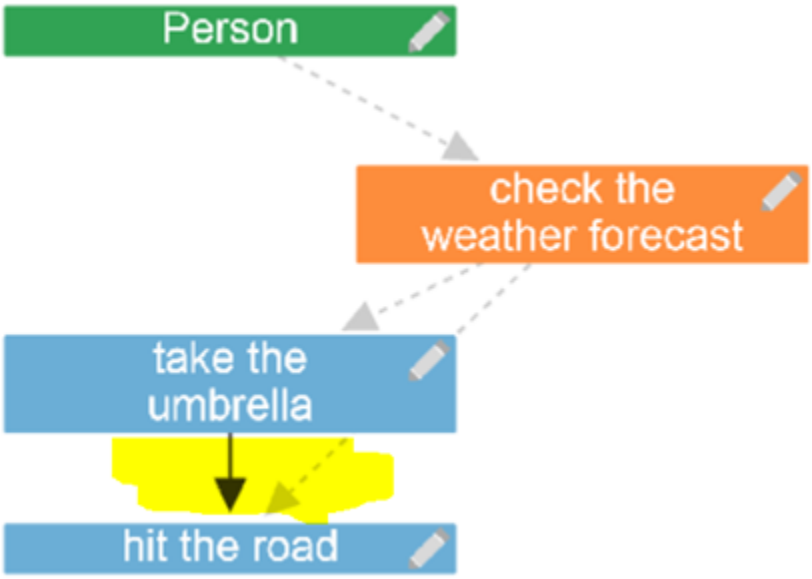
Let's try running the weather example above:

Notice that you get two traces (either the umbrella or the sunglasses), only one of which is shown below for the purpose of illustrating the GUI depiction of precedence (FOLLOWS) and composition (IN).

Color coding is used in both the code and the GUI (but not in this tutorial) to indicate ROOT, composite and atomic events. *The color coding is a function of the development environment and not part of the MP standard.*

The arrows shown on the GUI also indicate composition (IN indicated by dashed, light arrows) and precedence (FOLLOWS indicated by solid, dark arrows) relationships. It should be noted that the default presentation on the GUI shows composition and precedence arrows superimposed. The event blocks can be dragged on the GUI to separate the arrows as illustrated below:





Note the : (event pattern); construction when defining the ROOT Person and the definition of the composite event Check_the_weather_forecast. In this example (event patern) is very simple – this same syntax is used to define much more complicated sets of events as will be seen in the expanding examples in this tutorial.

Encapsulation, Conditional and Optional Events

In this section we will cover Encapsulation (composition) of events, conditional and optional events.

Now let's try running with some conditionals. Notice that the conditions look just like events in terms of syntax. If this seems confusing, try thinking of the conditionals (It_will_rain and No_rain) as outcomes (events) resulting from checking the weather.

Precedence And Debugging Our First Syntax Error

Precedence is the ordering of events in the model. In MP syntax, a space between two events creates precedence as in the example below.

Naming conventions for different model elements are also illustrated below. Consult the MP Style guide for more information about naming conventions. Recall that MP is case sensitive.

Example 2 Weather With Conditionals

/* This the “weather example” with conditionals details*/

ROOT Person**: (Check_the_weather_forecast);**

Check_the_weather_forecast

(It_Will_Rain Take_the_umbrella |

No_Rain Take_the_sunglasses)

/*The syntax error is fixed by putting in the : after Check_the_weather_forecast*/

Composite and concurrent events

Now, “Attend_class” may be a good opportunity to define a composite event and to bring some concurrent (parallel) events in the picture. For example, if Student is the name of root event (the main actor in your model) then you may define a composite event “Attend_class’ as follows.

This is new example, so no highlighting is shown.

Example 3 Attend Class Starter (composition, concurrence and precedence)

ROOT Student: Attend_class;

/* this is a composite event definition

events separated by spaces indicates precedence-

meaning that they happen exactly in the order specified */

Attend_class: Find_a_good_seat

Plug_in_laptop

/* activities in {,, } may happen in parallel – concurrent events*/

{ Listen_to_instructor, Make_notes,

Check_your_email_during_a_pause }

Leave_the_classroom;

Notice that you got two traces: one in which the student did nothing (did not attend class) and one in which all activities inside Attend_class were completed.

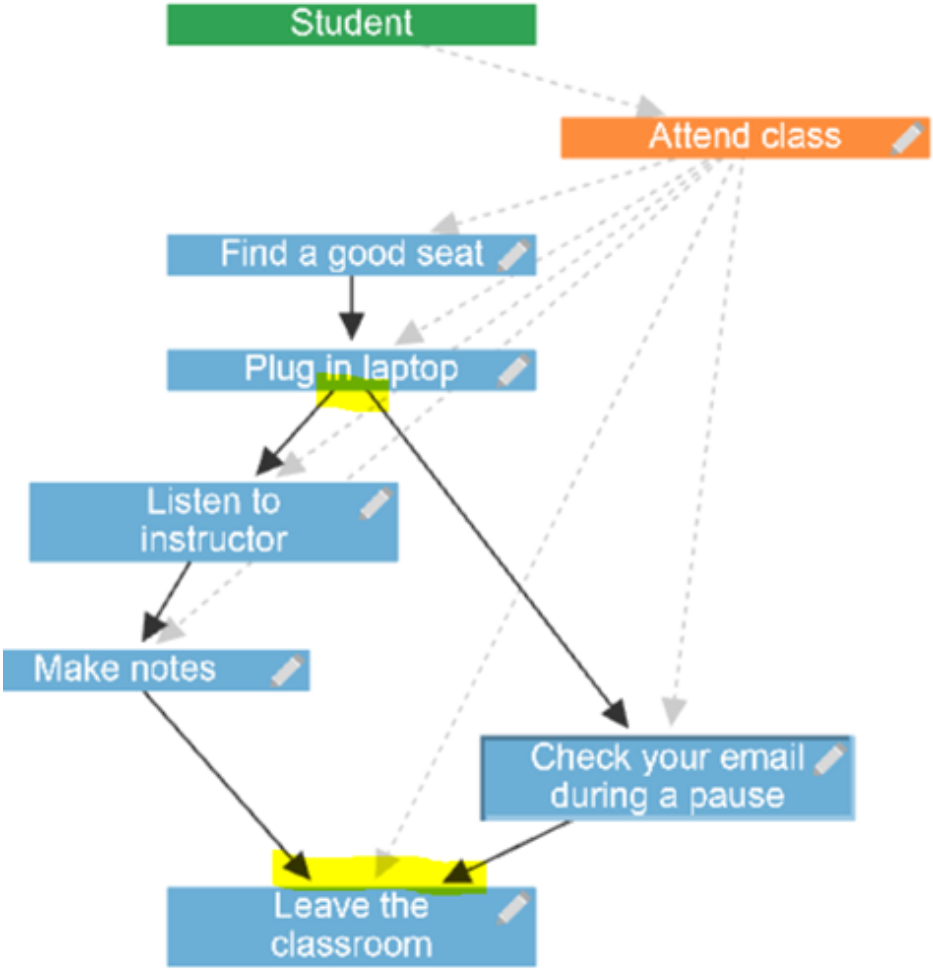
Also of note is the syntactic significance of the commas in the concurrent events.

Concurrence and precedence can be mixed, as in:

{ Listen_to_instructor Make_notes,

Check_your_email_during_a_pause }

Make_notes always follows Listen_to_instructor which can be concurrent with Check_your_email_during_a_pause. Remember, spaces indicate precedence and commas indicate concurrence. To be able to see the concurrence clearly on the GUI, pulling to the side to separate the IN from the FOLLOWS is needed. Note as shown by the highlighting that the parallel paths (concurrence) is now shown clearly on the GUI.



Optional Events

Now suppose we want to model the possibility that after class you might want to proceed to the library to study more, but you also might want to just go home. It also might be that you skip class and just go the library. So, we add the event Go_to_library to the prior example (. Note that Go_to_library is made optional by the [ ] syntax and that it is added as an event separate from Attend_class. In this example precedence between the two upper level events is set with Attend_class always before Go_to_library which is indicated by the space between them.

Notice that this is a different way of framing the problem than many other programming languages which would use an explicit if-then-else structure and would have to explicitly specify the conditions under which you went to the library (or not). In MP the focus is on the possibility of this behavior happening rather than understanding the exact conditions under which it might happen (which is, of course, important at the end of the day and can be specified in MP, but not right now). If I really want to specify reasoning for deciding library or not library, conditionals, as explained in Example 2 Weather With Conditionals, would be appropriate.

Example 4 Optional Events

ROOT Student**:** Attend_class **[**Go_to_library**];**

/ this is a composite event definition */*

Attend_class: Find_a_good_seat

Plug_in_laptop

/ these activities may happen in parallel */*

{ Listen_to_instructor,

Make_notes,

Check_your_email_during_a_pause }

Leave_the_classroom;

This generates four traces total: no class, library only, class+library, class only. Notice that class and library happen in a sequence (i.e. precedence), but are independent events. In contrast, all the elements inside the composite event Attend_class *always* happen (whever Attend_class happens). The reason for generating only a single trace for Attend_class despite showing three events is that the { } confers the meaning that order does not matter, so there is only one trace necessary.

Now let's add an optional event Ask_prof_a_question before leaving the classroom.

Example 5 More Optional Events

ROOT Student**:** Attend_class **[**Go_to_library**];**

/* this is a composite event definition */

Attend_class: Find_a_good_seat

Plug_in_laptop

/* these activities may happen in parallel */

{ Listen_to_instructor,

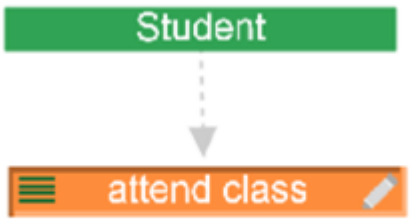
Make_notes,

Check_your_email_during_a_pause }

[Ask_prof_a_question]

Leave_the_classroom;

The placement outside { } indicates that you can't interupt the Prof to ask your question.



.

Figure 8 Example 5 Trace 1

Attend_class is collapsed (by clicking on the pencil icon) to hide the atomic events. You can see this on the GUI by the bars on the left of the Attend_class.

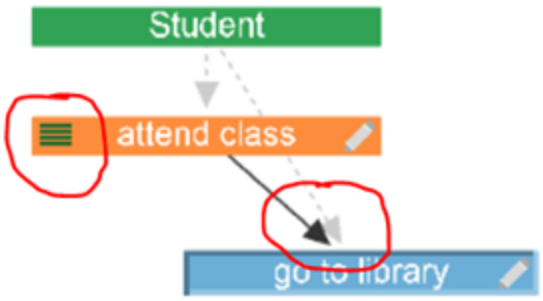


Figure 9 Example 5 Trace 2

In this sequence of events (trace) you went to the library after class. Dragging the Go_to_library on the GUI allows you to see that library follows attending class (solid arrow).

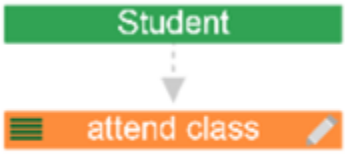


Figure 10 Example 5 Trace 3

The collapsed Attend_class makes this look just like trace #1 because the two alternatives within Attend_class (ask question or don't) are hidden. Using the event collapse on the GUI can be a good way to reduce clutter, but just be aware of what you are looking at.

Trace 4 is similar to trace #2 shown above.

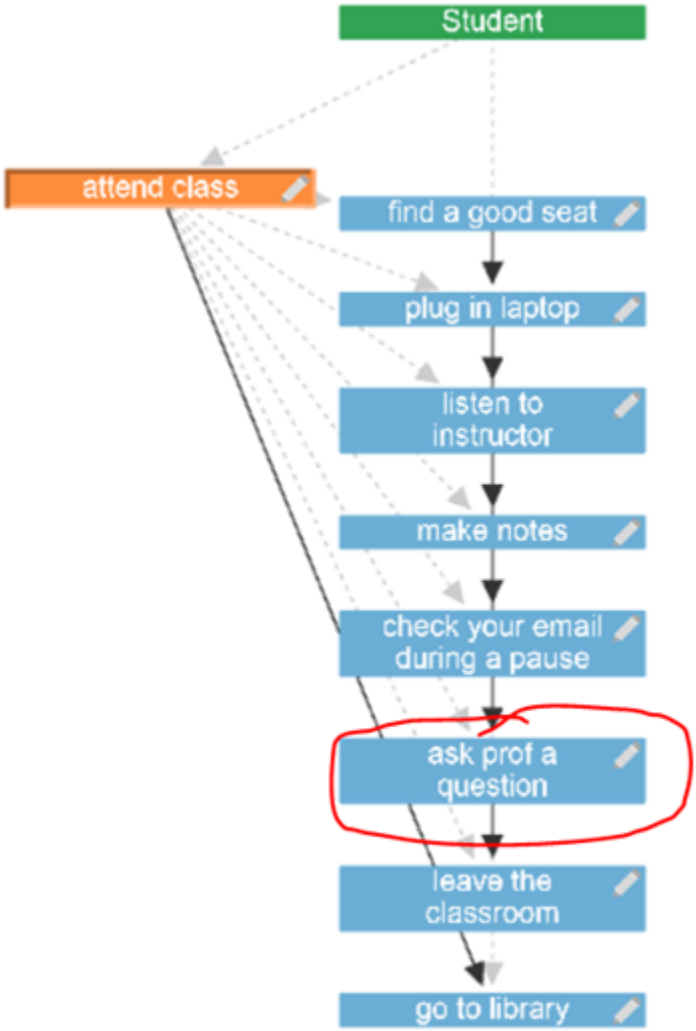


Figure 11 Example 5 trace 4

Simplify Visuals Through Decomposition

As you can see this example has a lot of different atomic events shown in each trace, which can easily become hard to digest visually. An alternative way to think about the problem by using more decomposition can help with this. The decomposition in this example is specifically chosen in a way which keeps all non-optional events together inside a single parent event. Now, when the parent nodes are collapsed you can easily see that the four event traces come from the inclusion/exclusion of the two optional events (library and question) with all other things being equal.

ROOT Student: Attend_class [Go_to_library];

/* this is a composite event definition */

Attend_class:

get_ready lecture [Ask_prof_a_question]

Leave_the_classroom;

get_ready: Find_a_good_seat Plug_in_laptop;

lecture:

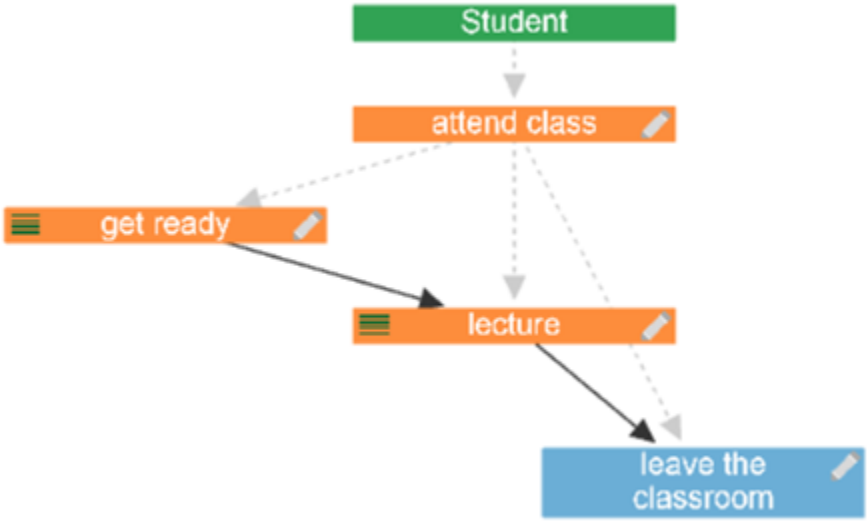
/* these activities may happen in parallel */

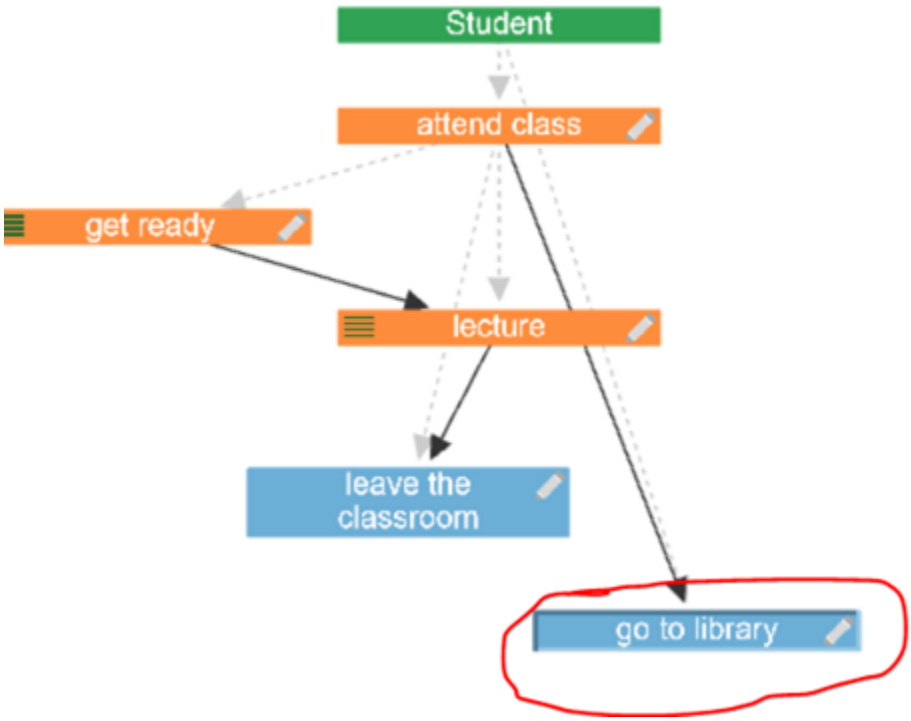
{ Listen_to_instructor,

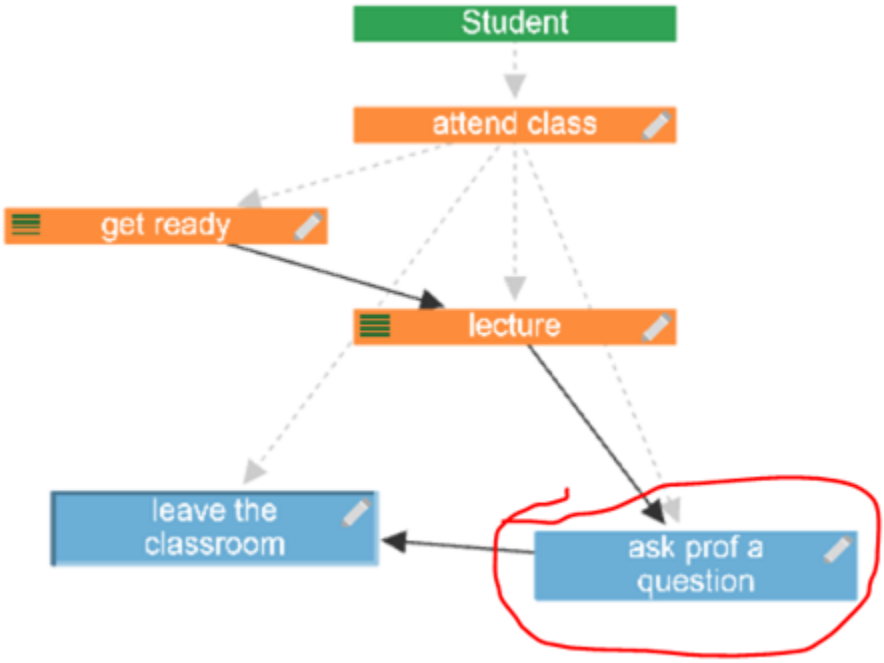
Make_notes,

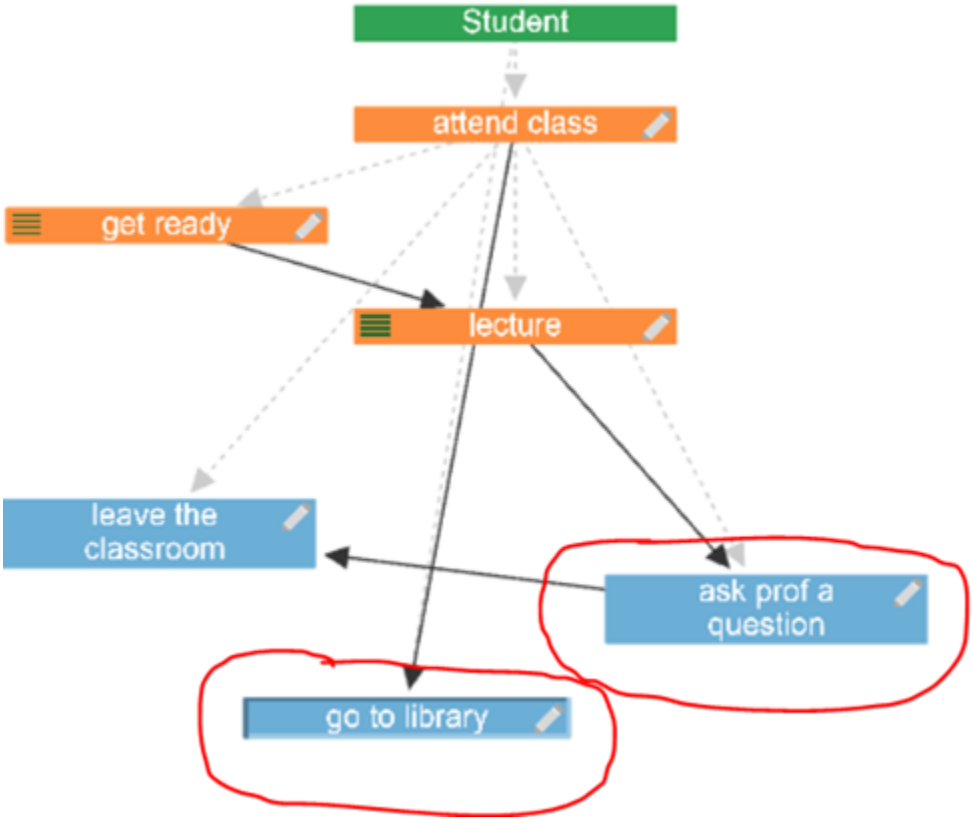
Check_your_email_during_a_pause };

Example 6 Attend_class with more decomposition




Student
attend class
get ready
lecture
leave the classroom
go to library


Student
attend class
get ready
lecture
leave the classroom
ask prof a question



Now let us say that sometimes you go Visit_with_friends instead of going to the library. This is included as an alternative event as shown. Note that the ( ) syntax must be used, not just the |, to indicate "or". This allows sequences of events to be included in the or construct. Also note that since the first line

is getting a bit long, you can just continue to the next line with no special continuation character. *Indentation is good coding practice for readability*, but is NOT enforced by MP.

Example 7 Optional And Alternative and Line Continuation

ROOT Student**:** Attend_class

[(Go_to_library|Visit_with_friends)];

/* this is a composite event definition */

Attend_class: Find_a_good_seat

Plug_in_laptop

/* these activities may happen in parallel */

{ Listen_to_instructor,

Make_notes,

Check_your_email_during_a_pause }

[Ask_prof_a_question]

Leave_the_classroom;

You now have another option (Visit_with_friends) and there is also Visit_with_friends option if you do not attend class since library and friends are independent (and mutually exclusive in this case) events.

5. Iterating With MP (Repetitive Event Patterns)