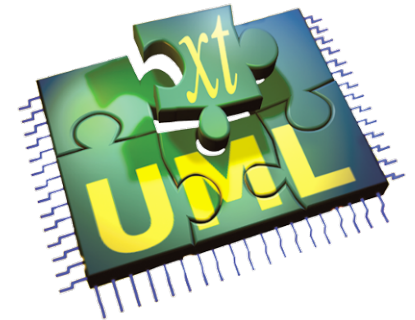
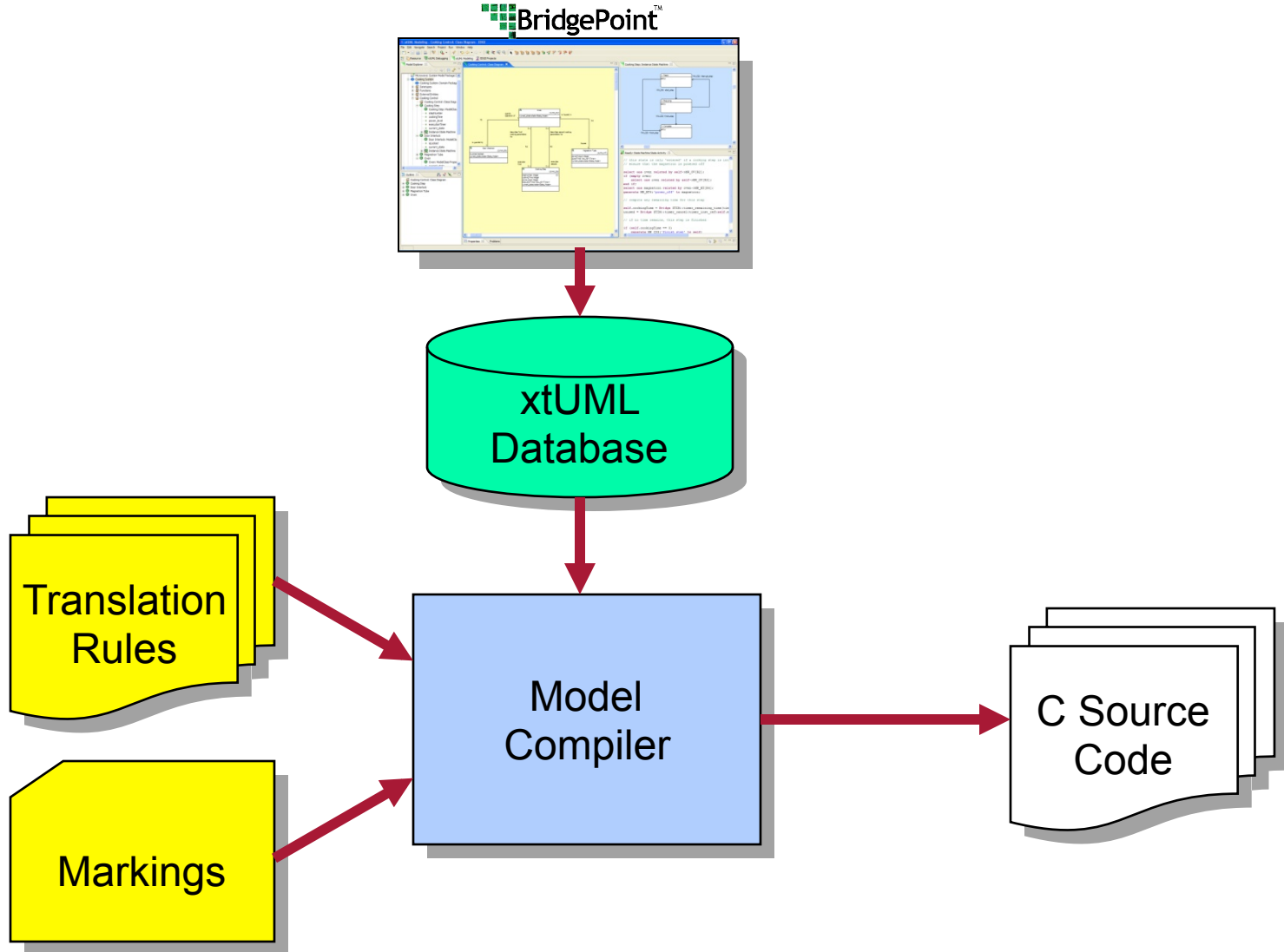


# The xtUML method – Code Generation

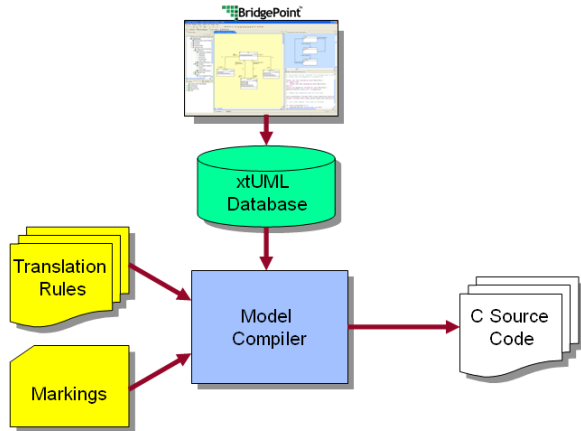
- ◆ **Analysis** – questioning, thinking, sketching...
  - Descriptive UML diagrams
    - use case, sequence, ...
- ◆ **Executable Modeling** – formalizing the analysis:
  - Component Diagrams (partitioning/interfaces)
  - Class Diagrams (data)
  - State Machines (control)
  - Activities (processing)
- ◆ **Verification**
  - Interpretive Model Execution
- ◆ **Code generation**
  - **Template and Rule-Based Translation**



# Code Generation



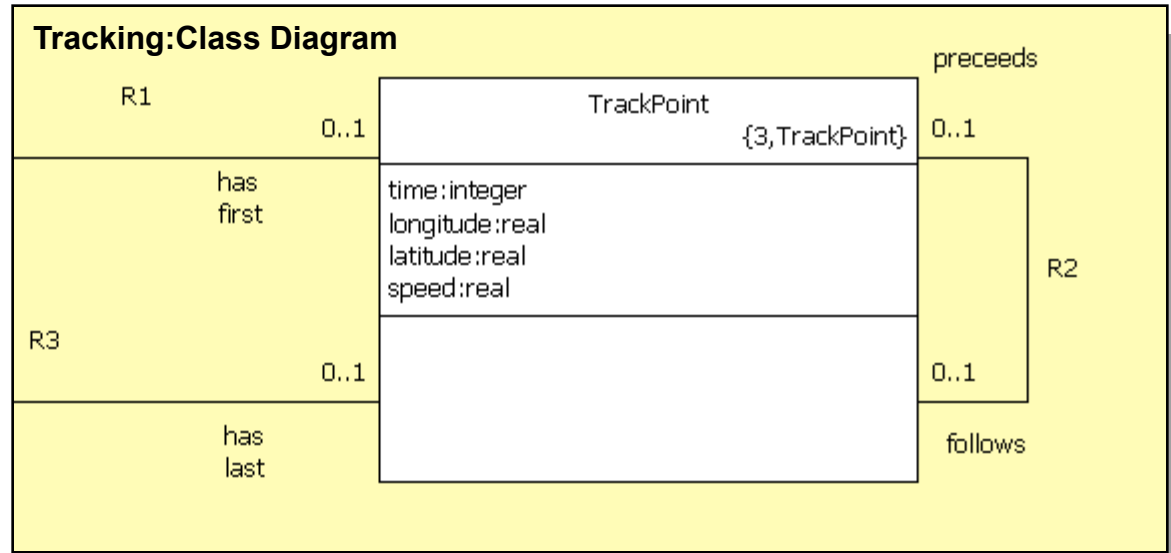
# Generating Classes



Tracking\_Trackpoint\_class.h

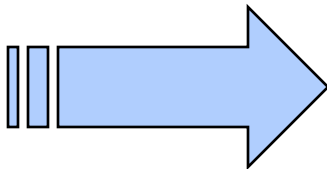
```

struct Tracking_TrackPoint {
  /* application analysis class attributes */
  i_t time; /* - time */
  r4_t longitude; /* - longitude */
  r4_t latitude; /* - latitude */
  r4_t speed; /* - speed */
  /* relationship storage */
  /* Note: No storage needed for TrackPoint->TrackLog[R1] */
  Tracking_TrackPoint * mc_TrackPoint_R2_follows;
  Tracking_TrackPoint * mc_TrackPoint_R2_preceeds;
  /* Note: No storage needed for TrackPoint->TrackLog[R3] */
};
    
```



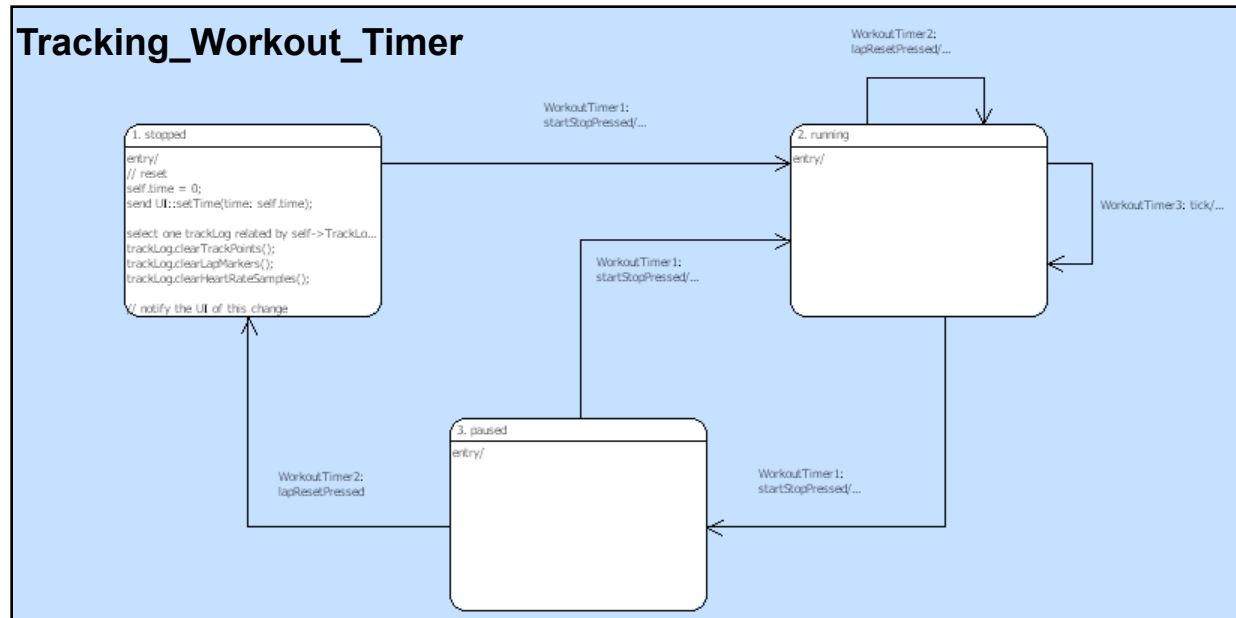
# Translation Rules

```
struct <class name>_s {
    <attr1 type>    <attr1 name>; /* <attr1 description> */
    <attr2 type>    <attr2 name>; /* <attr2 description> */
    <attr3 type>    <attr3 name>; /* <attr3 description> */
    . . .
    /* Association storage */
    <ref1 class name>_s * < ref1 class name >_<assoc1 number>;
    <ref2 class name>_s * < ref2 class name >_<assoc2 number>;
    . . .
    /* State machine current state */
    StateNumber_t    current_state;
};
```



```
struct Tracking_TrackPoint {
    /* application analysis class attributes */
    i_t time; /* - time */
    r4_t longitude; /* - longitude */
    r4_t latitude; /* - latitude */
    r4_t speed; /* - speed */
    /* relationship storage */
    /* Note: No storage needed for TrackPoint->TrackLog[R1] */
    Tracking_TrackPoint * mc_TrackPoint_R2_follows;
    Tracking_TrackPoint * mc_TrackPoint_R2_precedes;
    /* Note: No storage needed for TrackPoint->TrackLog[R3] */
};
```

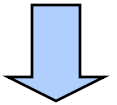
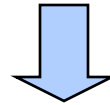
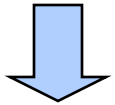
# State Machine Generation



```
state = instance->current_state;
next_state = Tracking_Workout_Timer_StateEventMatrix[ state ]
[ event_number ];
/* Update state and execute the state action */
instance->current_state = next_state;
( *Tracking_Workout_Timer_Actions[ next_state ] )( instance, eventData );
...
```

## Translation Rules: Event Dispatch

```
...  
state = instance->current_state;  
next_state = <class name>_StateEventMatrix[ state ][ event_number ];  
/* Update state and execute the state action */  
instance->current_state = next_state;  
( *<class name>_Actions[ next_state ] )( instance, eventData );  
...
```



```
...  
state = instance->current_state;  
next_state = Tracking_Workout_Timer_StateEventMatrix[ state ]  
[ event_number ];  
/* Update state and execute the state action */  
instance->current_state = next_state;  
( *Tracking_Workout_Timer_Actions[ next_state ] )( instance, eventData );  
...
```

## Generated Code structure

- ◆ StateEventMatrix contains next state for each current state and input event.

```
Next_state = <class name>_StateEventMatrix[ state ][ event_number ];
```

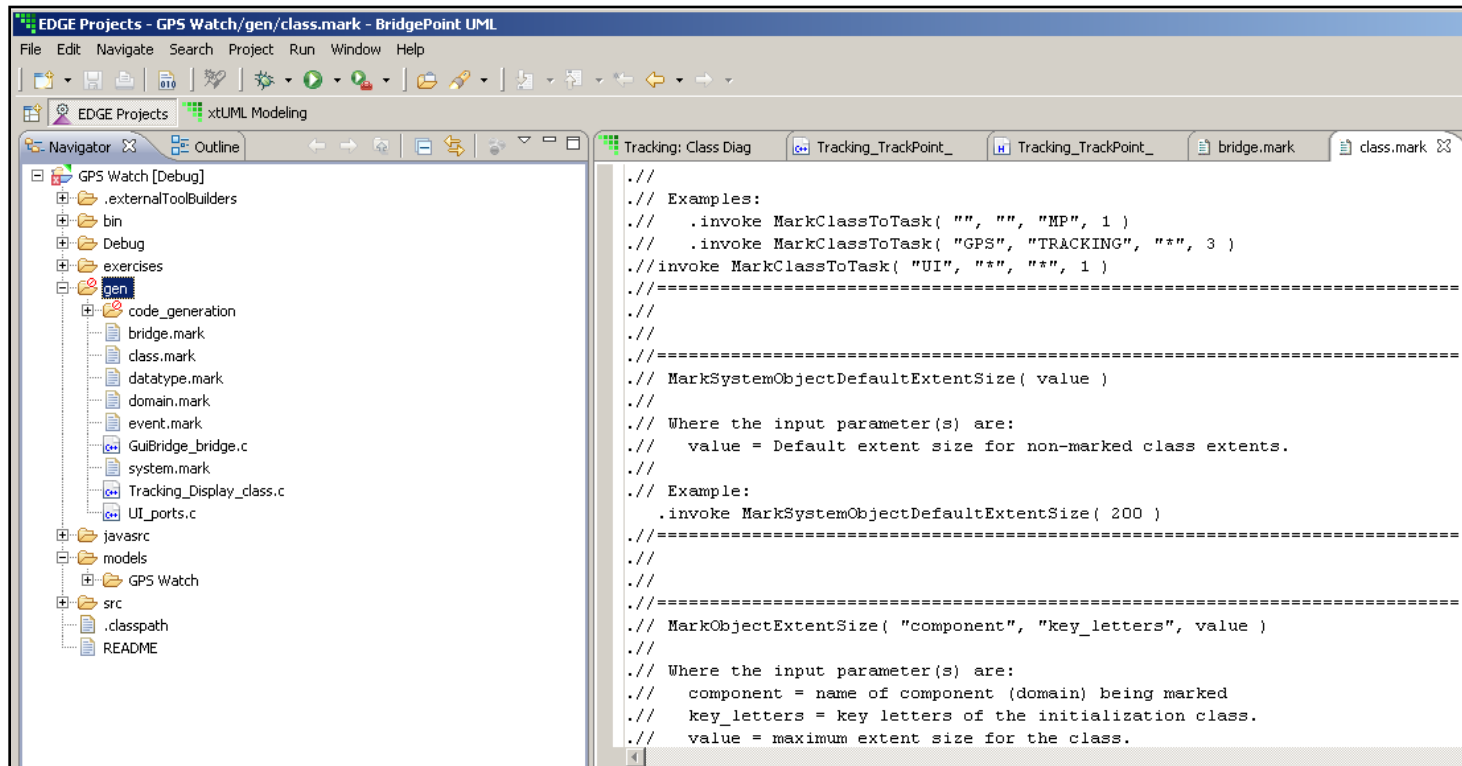
- ◆ Actions is an array of function pointers to the generated action code for the state itself.

```
( *<class name>_Actions[ next_state ] )( instance, eventData );
```

- ◆ One Procedure for each state machine, and they all are essentially the same – Only the class name is unique.

# Markings

- ◆ Contained in the gen folder.
- ◆ .mark files control details of the code generation.
- ◆ 6 mark files available.  
System, domain, class, event, bridge, datatype



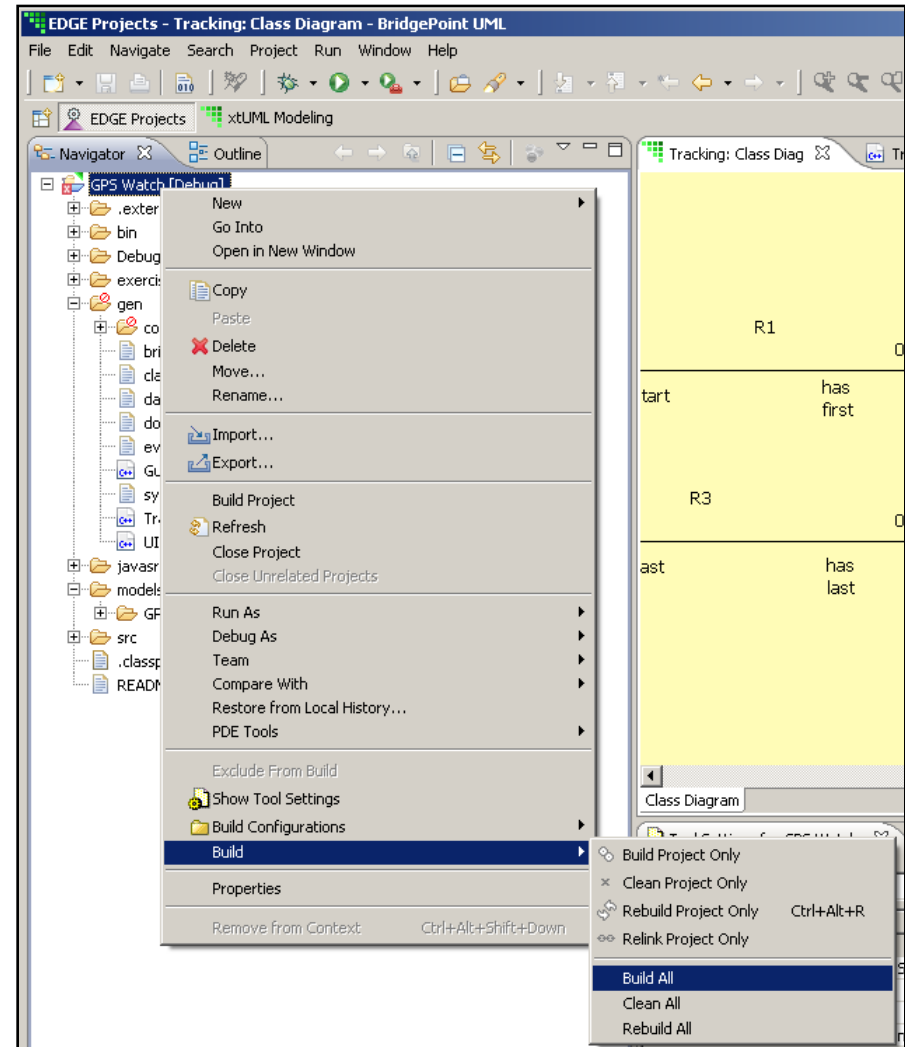


# Useful Markings

- ◆ **MarkActionStatementTracingOn()** in **domain.mark**  
MarkActionStatementTracingOn is used to enable the generation of trace macros into the generated code that will output run-time trace statements of the Object Action Language statements executed during the run.
- ◆ **MarkInitializationFunction( "comp", "fname" )** in **domain.mark**  
Designate a function to serve as an initialization function in a domain.
- ◆ **TagDataTypePrecision( "domain", "dt\_name", "tagged\_name", "initial\_value" )** in **datatype.mark**  
To indicate the 'precision' of a user defined data type. (e.g. double)

# Performing Code Generation

- ◆ Use the C/C++ Perspective in Eclipse and Build Project.
- ◆ Batch generation: `xtumlc_gen_erate`
  - `-import xtUML_file`
  - `-nopersist`
  - `-v verbosity`
  - `-f output_filename`



# Document Generation

- ◆ Generate HTML Documentation from your models.

The screenshot displays the xtUML Modeling application interface. The 'Model Explorer' on the left shows a context menu with 'Create documentation' selected. The main window displays the generated HTML documentation for a system named 'GPS Watch'. The documentation includes a title, a description, and a section for 'System-Level Component Packages'. Under the 'System' sub-section, there is a code block for a component package and a diagram labeled 'Figure 1. System Component Package Diagram'. The diagram shows a component package 'UI' with a sub-component 'UI' connected to a circle representing a port.

```
// ##### START OAL_1 ##### // always initialize data self.currentLocation.longitude = 0.0; self.currentLocation.latitude = 0.0; self.currentLocation.speed = 0.0; self.interval = 2000000; LOG:LogInfo(message: "Location listener registered."); // start timer based on requested interval create event instance timeout of GPS3(timeout()) to self: self.timer = TIM:timer_start_recurring(event_inst: timeout, microseconds: self.interval); // ##### END OAL_1 ##### //
```

Figure 1. System Component Package Diagram

```
graph TD
    subgraph UI
        UI2[UI]
    end
    UI --- UI3((UI))
```