SynthML instructions

September 4, 2025

1 Create a new SynthML model

Let us first open the SynthML environment within UML Designer. A snapshot of the main screen is shown in Fig. 1, with the main components highlighted. In order to create a new UML model,

1. Right click in the 'Model Explorer' View → New → UML Project, and give it a name.
2. Right click on the project → Viewpoints Selection → Check 'A_SynthML'. This enables all the SynthML features, such as the addition of guards and effects to opaque behaviors.

The model will have a UML file by default, called `model.uml`. Within it, there is a model instance denoted `<Model> NewModel`. In the following, we refer to this model instance as the 'model', unless otherwise specified.

1.1 Create a Class

A UML class represents a (sub)system that has state and behaviors. One of the behaviors of an *active* UML class must be appointed the *classifier* behavior of that class. An active class starts its classifier behavior (*e.g.,* an activity) as a direct consequence of its creation. There are several ways of creating a class in UML Designer. From the 'Model Explorer' view,

1. Right click on the model → New Child → Packaged Element[1] → Class. You may want to rename it with a meaningful name.
2. We only support active classes: to make a class active, look in the 'Properties' view, 'Advanced' tab, and check the 'Active' checkbox.

[1] A class can also be added as an 'Owned Type' by right clicking on the model → Owned Type → Class. The UML metamodel treats both owned types and packaged elements as 'packageable elements': it makes no practical difference for the creation of a SynthML model. For consistency, it is advised to use the packaged element option.

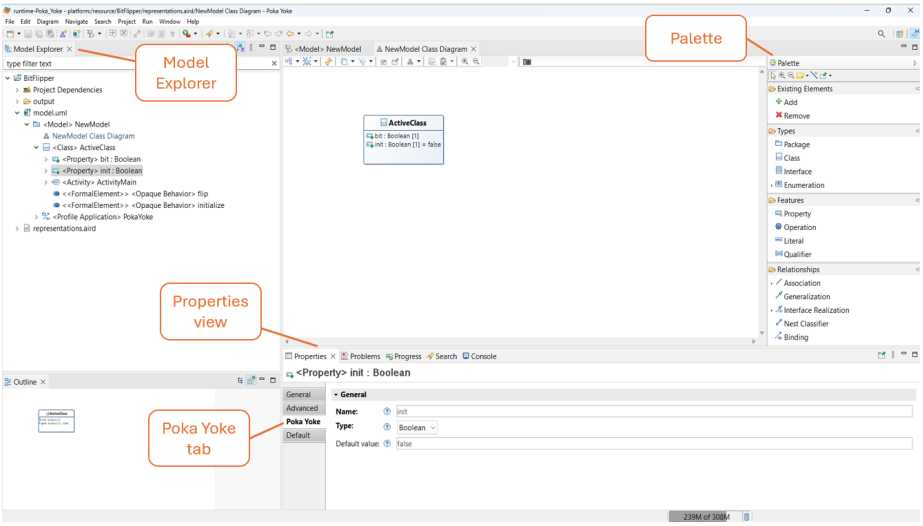

Figure 1: A screenshot of the SynthML environment.

Graphical Approach. Alternatively, you may create a class from the 'Class Diagram' view. To this end, you may

1. Right click on the model → New Representation → Class Diagram.
2. A new empty class diagram opens, with some small text mentioning 'Your diagram is empty. You can either [...]'. Double click on that text. If the model instance already contains a class, *e.g.,* from the previous steps, then this class will be visualized in the class diagram. Otherwise, the class diagram will be blank.
3. On the right hand side, the 'Palette' view contains several folders: 'Existing elements', 'Types', 'Features', and 'Relationships'.
4. To create a class, simply click on the 'Class' element (within 'Types') and drag it to the diagram.

1.2 Create a Property

Properties represent the variables of a model. The supported types are booleans, enumerations and ranged integers (besides composite data types, see §1.3). To create a boolean-typed property:

1. Right click on the Class → New Child → Owned Attribute → Property.
2. In the 'Properties' view, in the 'SynthML' tab, you can select 'Boolean' as the type, and set a default value for the type. To create integers and enumerations, you need to define them beforehand, as explained in the following.

To create an enumeration-typed property, first we need to define the enumeration. To do so,

1. Right click on the model → New Child → Owned Type → Enumeration. Alternatively, you could drag-and-drop an 'Enumeration' from the 'Palette' view on the right.

2. Right click on the Enumeration → New Child → Owned Literal → Enumeration Literal. Alternatively, you could hover over the enumeration with your mouse in the class diagram, after which some options appear, from which you can select Literal.

After the completion of these steps, we can create a new property of type Enumeration (with the given enumeration name).

Similarly, to create an integer-typed property, first we need to define the ranged integer type. To do so,

1. Right click on the model → New Child → Owned Type → Primitive Type. Alternatively, you could add a Primitive Type from the 'Palette' view, by clicking on the tiny arrow on the left of 'Enumeration', which reveals 'PrimitiveType', allowing it to be dragged-and-dropped to the class diagram.

2. In the 'Properties' view, within the 'SynthML' tab, select 'Super type: Integer'. Give it a name and write down the minimum and maximum values. Note that two constraints named 'min' and 'max' will appear in the 'Model Explorer' view.

After the completion of these steps, we can create a new property of this ranged integer type.

Both enumerations and primitive types can also be added as a 'Packaged Element' by right clicking on the model → Packaged Element → Enumeration (or Primitive Type). The UML metamodel treats both owned types and packaged elements as 'packageable elements', which are the elements considered by SynthML, so it makes no practical difference for the creation of a SynthML model. Since enumerations and primitive types are usually considered as types, it is advised to use the owned type option.

Graphical Approach. Alternatively, you may create a property from the 'Class Diagram' view. To this end,

1. From the 'Palette' view, simply click on the Property element (within Features) and drag it inside a class. A popup window will open: fill in the Name, then click OK.

2. Double click on the property, in the 'Properties' view, 'SynthML' tab, select the property type from the drop-down menu.

Note that you still need to create an enumeration and integer type via the standard approach before you can create an integer or enumeration property.

1.3 Create a (Composite) Data Type

In UML diagrams, data types are model elements that define data values. You typically use data types to represent primitive types, such as integer types, and user-defined data types. To avoid confusion among types, we refer to *composite* data types to elements of type `DataType` who are *not* also enumerations or primitive types. Composite data types are used to represent more abstract types, *e.g.,* a robot. To create a composite data type,

1. Right click on the model → New Child → Owned Type[2]→ Data Type.

Note that composite data types can have properties of type boolean, integer, enumeration, or of other composite data type – as long as this does not introduce a cyclic structure.

Graphical Approach. Alternatively, you may create a composite data type from the 'Class Diagram' view. To this end, you may

1. In the 'Palette' view, click on the tiny arrow located at the left of Enumeration. A drop-down menu will open, revealing 'DataType'.
2. Click on 'DataType', and drag it to the class diagram. A pop up window will appear, where you can change its name; finally, click OK.

1.4 Create an Abstract Activity

Activities can either be abstract (in order to be synthesized) or concrete (*i.e.,* containing nodes and control flows created by the user). Let us consider abstract activities.

1. Right click on the Class → New Child → Classifier Behavior → Activity.
2. By default, the activity is concrete (allowing it to contain user-defined nodes and control flows).
3. For synthesis purposes, an abstract activity is required: in the 'Properties' view, 'General' tab, check the 'Abstract' checkbox.

Note that a class can only have a single classifier behavior, and it *must* be an activity. So if you'd add an additional activity, this should be added as an 'Owned Behavior' instead.

To be able to synthesize a meaningful activity, we must define a precondition (see §1.4.1) and a postcondition (see §1.4.2), defining respectively the system states at the beginning and the end of the activity execution. Additionally, users can specify constraints, (see §1.7), defining additional requirements, and

[2]A composite data type can also be added as an 'Packaged Element' by right clicking on the model → Packaged Element → Data Type. The same consideration regarding the interchange of owned types and packaged elements mentioned earlier is valid. Since a composite data type is usually considered as a type, it is advised to use the owned type option.

occurrence constraints (see §1.8), limiting the number of times an element can be called by the activity. Finally, the synthesis procedure employs opaque behaviors (see §1.6) as building blocks to synthesize an activity.

Note that in UML Designer activities can also be added as a nested classifier. This is *not* allowed in a SynthML model.

1.4.1 Create Preconditions

1. Right click on the activity → New Child → Precondition.
2. Right click on the Precondition → New Child → Constraint.
3. Right click on the Constraint → New Child → Specification → Opaque Expression.
4. Open the opaque expression in the 'Properties' view. Delete the name, or leave it blank (this is optional, but may give nicer visualizations).
5. In the 'Properties' view, 'Default' tab, double click on the 'Body' field.
6. In the new window, write on the 'Value' field your precondition constraint. Click Add, then OK.

We follow similar steps for creating postconditions.

1.4.2 Create Postconditions

1. Right click on the activity → New Child → Postcondition.
2. Right click on the Postcondition → New Child → Constraint.
3. Right click on the Constraint → New Child → Specification → Opaque Expression.
4. Open the opaque expression in the 'Properties' view. Delete the name, or leave it blank (this is optional, but may give nicer visualizations).
5. In the 'Properties' view, 'Default' tab, double click on the 'Body' field.
6. In the new window, write on the 'Value' field the postcondition constraint. Click Add, then OK.

1.5 Create a Concrete Activity

The creation of a concrete activity starts similarly to the creation of an abstract activity:

1. Right click on the class → New Child → Classifier Behavior → Activity.

Notice that you do not check the 'Abstract' checkbox. Recall that a class can only have a single classifier behavior: if you have already added such a behavior (*e.g.,* an abstract activity), the concrete activity should be added as an 'Owned Behavior' instead.

A graphical representation of the activity is usually an activity diagram,

1. Right click on the activity → New Representation → Activity Diagram.

A concrete activity is a user-defined collection of nodes that are connected by control flows. It can thus be seen as a finite directed graph. A node can be a *control node* (initial, final, fork, join, decision, or merge node), be an *call behavior node* which either calls another activity of executes an action by calling an opaque behavior, or be an *opaque action node* containing a guard and zero or more effects, thereby essentially 'inlining' an action. So likewise to UML opaque behaviors, also opaque action nodes are either deterministic or non-deterministic, and either atomic or non-atomic. Concrete activities can be visualized via an activity diagram (see §1.4). Further, concrete activities can also be added as a nested classifier. This is *not* allowed in a SynthML model.

1.5.1 Parameterize a Concrete Activity

After having created an activity, parameterizing it can be achieved by

1. Open the activity diagram.
2. Left click once on the activity label in the top center to start direct editing.
3. Postfix the activity name with angle brackets containing the parameters, for example
   `MyActivity<ParamName1:Type1, ParamName2:Type2, ...>`.

The supported types are booleans, enumerations and ranged integers. See 1.5.5 to learn how to call such an activity.

1.5.2 Create a Control Node

After having created a concrete activity, adding a control node can be achieved by

1. Right click on the concrete activity → New Child → Node → Initial / Activity Final / Fork / Join / Decision / Merge node.

Note that the final node is denoted as 'Activity Final node'.

Graphical Approach. After having created the concrete activity, go to the activity diagram

1. From the 'Palette' view of the activity diagram, you can find the control nodes: clicking on the little arrow next to 'Initial node' opens a drop-down menu with all types of control nodes.

1.5.3 Create a Control Flow

After having created a concrete activity, adding a control flow can be achieved by

1. Right click on the concrete activity → New Child → Edge → Control Flow.
2. We now need to select the start and end node of this control flow. In the 'Properties' view, 'Default' tab, there are two columns, named 'Property' and 'Value'.
3. Look for the rows 'Source' and 'Target' under the 'Property' column. By clicking in the corresponding 'Value' column, you can select the source and target node of the control flow from a drop-down menu.

Graphical Approach. After having created the concrete activity, go to the activity diagram

1. From the 'Palette' view of the activity diagram, you can find the control flow. Click on the icon, then click on the node representing the source of the flow, then on the node representing the target of the flow.

Control flows support incoming and outgoing guards, which are found in the 'SynthML' tab of the 'Properties' view. An incoming guard specifies when a token can be placed on that control flow, whereas an outgoing guard specifies a condition for the removal of the token from the control flow. Both guards should be state predicates over the model properties, similarly to opaque action guards.

1.5.4 Create a Call Behavior Node

After having created a concrete activity, adding a call behavior node can be achieved by

1. Right click on the concrete activity → New Child → Node → Call Behavior Action node.
2. Within the 'Default' tab, the 'Behavior' row allows you to select which behavior to call with a drop-down menu, after having pressed the '...' button. This behavior can either be an activity, or an opaque behavior.

Graphical Approach. From the 'Palette' view of the activity diagram, you can find the 'Call Behavior' node under the 'Opaque Action' drop-down menu.

When a call behavior node represents a call to an activity, we may abstract the call by adding a guard and effects to the node itself, located in the 'SynthML' tab of the 'Properties' view. This is informally denoted as 'shadowed' call behavior. The execution of this node skips the actual call to the underlying activity, which is instead substituted by the the guard and effects. Note that

a shadowed call is allowed only when the called behavior is an activity, and forbidden when the called behavior is an opaque behavior.

1.5.5 Add arguments to a Call Behavior Node

After having created a call behavior node (see 1.5.4), passing arguments to a parameterized activity can be achieved by

1. Left click on the call behavior node.
2. In the 'Properties' view, 'SynthML' tab, under 'Arguments', add arguments as assignments, for example
   `ParamName1:=value1, ParamName2:=value2, ...`

Graphical Approach. After having created the call behavior node, go to the activity diagram

1. Left click once on the call behavior node label to start direct editing.
2. Postfix the name with angle brackets containing the arguments, for example
   `CalledActivity<ParamName1:=value1, ParamName2:=value2, ...>`.

Only literals and parameters of the calling activity can be passed as arguments. See 1.5.1 to learn how to add parameters to an activity.

1.5.6 Create an Opaque Action Node

After having created a concrete activity, adding an opaque action node can be achieved by

1. Right click on the concrete activity → New Child → Node → Opaque Action node.
2. In the 'Properties' view, 'SynthML' tab, you can add guards and effects for the opaque action.

Graphical Approach. From the 'Palette' view of the activity diagram, you can directly find the 'Opaque Action' node.

1.6 Create an Opaque Behavior

Opaque behaviors represent actions that activities can perform. Opaque behaviors can be atomic (meaning that the system cannot perform any other operation during the execution of an atomic action) or non-atomic, and deterministic (i.e., having at most one effect) or non-deterministic (i.e., having multiple effects, of which only one is chosen to be executed non-deterministically). For activity synthesis purposes, the user needs to create the actions to be used by the to-be-synthesized activities, as opaque behaviors.

1. Right click on the Class → New Child → Owned Behavior → Opaque Behavior.

2. For atomic actions, in the 'Properties' view, 'SynthML' tab, check the 'Atomic' checkbox.

3. In the 'Properties' view, 'SynthML' tab, in the 'Guard' field, the action guards can be specified (*i.e.,* the conditions under which this action can be performed).

4. In the 'Properties' view, 'SynthML' tab, in the 'Effects' field, the action effects can be specified (*i.e.,* the variable assignments that describe how performing the action changes the valuation of these variables).

5. Non-deterministic effects are separated by three tilde ~ signs. For example, an effect field `val:=true ~~~ val:=false` means that the variable `val` will be non-deterministically assigned *either* true *or* false as a consequence of performing the action. Note that the user has no control over which effect is picked by the execution of the action.

1.7 Create Constraints

For activity synthesis, in addition to abstract activities with pre- and postconditions, and opaque behaviors, also requirements can be specified, to which all synthesized activities must adhere to. Such requirements are specified in UML as class constraints. To create a class constraint,

1. Right click on the Class → New Child → Owned Rule → Constraint.

2. Right click on the Constraint → New Child → Specification → Opaque Expression.

3. Open the opaque expression in the 'Properties' view. Delete the name, or leave it blank (this is optional, but may give nicer visualizations).

4. In the 'Properties' view, 'Default' tab, double click on the 'Body' field. The expression body can be several things:

   - A state invariant, *e.g.,* `field1 and not field2`.
   - A state/event exclusion invariant, *e.g.,* `action1 needs field1 and not field2`, or `field1 and not field2 disables action1`.

1.8 Create Occurrence Constraints

Occurrence constraints define the amount of times a specific UML element (*e.g.,* activity or opaque behavior) can be called during the execution of an activity. As such, these are modeled as *interval* constraints in UML Designer. We might think of a specific action that we need to use (*i.e.,* the minimum of the occurrence constraint is greater or equal than one) and we do not want to use it

more than three times (*i.e.,* the maximum of the occurrence constraint is set to three). Let us assume we want to add an occurrence constraint to an activity. From the 'Model Explorer' view,

1. Right click on the abstract activity → New Child → Owned Rule → Interval Constraint. You may want to change the name to something representative of the constraint. In the 'Properties' view, click on the 'Default' tab and add an element in the 'Constrained Element' field. To do so, just click in the empty field of the corresponding 'Value' column, and a small icon with three dots appears. Click on that icon, and select the constrained elements from the list; click Add, and then OK.

2. We now need to define the minimum and maximum values for the occurrence constraint. Right click on the interval constraint → New Child → Specification → Interval. Again, you may want to rename it in a meaningful way.

3. We now insert a minimum and maximum value for the occurrence limits. To this end, we add a `Literal Integer` *at the model level*. Right click on the Model → New Child → Packaged Element → Literal Integer. For visualization purposes, erase the name, and fill in the 'Value' field the value you need. Notice that you may need two literal integers, one for the minimum and one for the maximum occurrence limit.

4. Click on the interval specification we have just defined. Within the 'Properties' view, click on the 'Advanced' tab, there are the 'Max' and 'Min' fields. Click on the three-dots icon and select the maximum and minimum values represented by the two literal integers defined in the previous step.

1.9 Model Validation

In order to use an UML model for simulation or synthesis, the model must be valid, otherwise an error will be given. To validate an UML model, open its class diagram (see §1.1), right click in an empty place of the diagram and select 'Validate diagram'. Any problems will be shown in the 'Problems' tab (which is next to the 'Properties' tab).

1.10 Perform Synthesis

In case the UML model has abstract activities, then concrete activities can automatically be computed for them by means of activity synthesis. This requires that the UML model is valid. Activity synthesis can be performed by the following steps:

1. Right click on the `model.uml` → Perform Synthesis. A new `output` folder is created.

2. Open the output folder, scroll down until the last `.uml` file, open it, open the model, open the class, open the activity.

3. For the activity diagram representation, right click on the activity → New Representation → Activity Diagram.

1.11 Simulation with Cameo

In order to find design problems in UML models, they can be simulated using Cameo. This requires translating the UML model to a (similar UML) model that Cameo can understand and simulate. This translation is automated, and can be performed via the steps described below. Note that *abstract* activities are not supported for simulation. You can however synthesize a concrete activity for every abstract activity by means of activity synthesis (see §1.10), and then translate the outputted UML model into a model that Cameo can simulate.

1. Right click on the `model.uml` → Translate UML to Cameo. A new `output` folder is created.
2. Open the output folder, open the folder called `model.uml` (*i.e.,* the name of the original UML file), and there you find a new UML file.
3. Open this UML file with the Cameo Systems Modeler, and simulate it.

2 Synthesis: Bit Flipper Example

To illustrate activity synthesis, let us now create a bit flipper model. In this model, the to-be-synthesized activity initializes a boolean variable non - deterministically, and it flips it if it is false; does not need to do anything otherwise. We can use the following steps:

1. Create a new UML project named BitFlipper.
2. Create an active class named ActiveClass.
3. Create two boolean properties, named `init` and `bit`. Set the default value of `init` to false, and leave an empty field (meaning no default value) for `bit`.
4. Create an abstract activity as the ActiveClass classifier behavior, with precondition `not init`, and postcondition `bit and init`.
5. Create an opaque behavior, named `initialize`, that models an initialization action, with guard `not init` and nondeterministic effects:
   ```
   init := true, bit := true
   ~~~
   init := true, bit:= false
   ```
   This action flags the initialization as done (`init` is true after the action) and non-deterministically instantiates `bit`.
6. Create a opaque behavior that models a bit flipper action, with guard `init` and effect `bit := not bit`, named `flip`.

7. Perform the synthesis. Figure 2 shows the activity diagram of the synthesized activity.

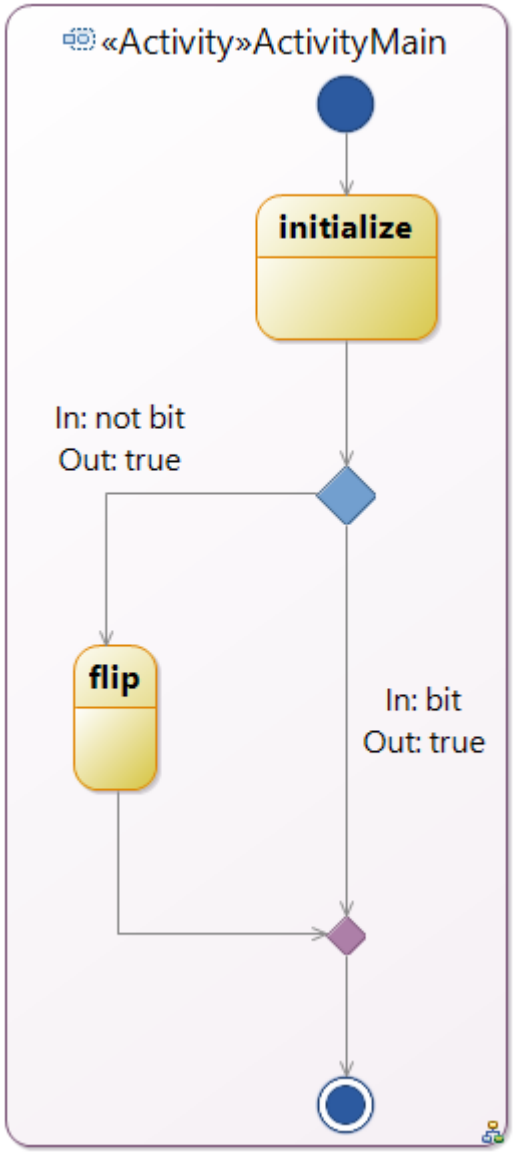

Figure 2: The synthesized activity diagram.