Using ColdFrame's TextUML

Simon Wright *simon@pushface.org*

April 4, 2019

Abstract

A worked example of the use of a textual form of UML to prepare translatable models.

Contents

1 Introduction

ColdFrame is an open-source code generator backend for use with UML tools, targeted at Ada.

Until recently (2019) the UML tool of choice has been ArgoUML; however, it hasn't been updated since 2015, aside from some test code.

The TextUML project is a Java-based tool to encode UML models in textual form. It goes beyond the aims of this project, in that it provides an action language. This means that the whole application can be written in TextUML and executable code can be generated from it.

ColdFrame doesn't go as far as this: it generates a framework, which can call up user code in the form of separate subprograms. Recently, it's been made possible to include some user code (in Ada) in the model.

This document has been generated using nuweb.py, with conversion to PDF via TeX Live.

2 Worked Example: Simple Buttons

The syntax of ColdFrame's version of TextUML is reproduced in Section 4.

A TextUML model can contain multiple domains. It acts only as a holder; its name has no significance. This is a 'file' scrap (the introductory `@o`), as encoded in the source web, which results in the other scraps in the document being 'tangled' into the file indicated.

```
@o textuml.tuml @{
(*
  This is a model comment, which appears before the element concerned
  and will be included in the output.
*)
/* This is a textual comment, which will be ignored. */
model TextUML\_Demonstration;
   @< The domains @>
end.
@}
```

It gets 'woven' as

"textuml.tuml" 2≡

```
(*
  This is a model comment, which appears before the element concerned
  and will be included in the output.
*)
/* This is a textual comment, which will be ignored. */
model TextUML_Demonstration;
   ⟨ The domains 3a ⟩
end.
◇
```

If you were generating the TextUML file by hand, you'd write documentation as model comments. Here, they've been expressed in the document instead, to improve readability.

The `Simple_Buttons` domain (the only one in this model) is intended for demonstrating ColdFrame's use in Ravenscar systems.

Because sampler boards have very few buttons and user-accessible LEDs, the design is very restricted. A Button can receive a short push (less than a quarter of a second) or a long push; after a short push, it's 'set' for 5 seconds; after a long push, it's 'set' until another push. A Button can be wired to control one or more LEDs; an LED can be wired to be controlled by one or more Buttons.

In TextUML, a domain is a package with the *annotation* `[domain]` ("annotation" is the TextUML word for *stereotype*). Other ways of decorating the model elements are *modifiers*, which are (reserved) keywords in the syntax; for example, you could specify that an attribute is identifying either by using the modifier `id` or the annotation `[\id]` (the backslash is removed during processing, but allows you to use otherwise-reserved identifiers).

A domain package can have nested packages, whose contents are incorporated directly into the domain.

⟨ *The domains* 3a ⟩ ≡

```
[domain]
package Simple_Buttons;
    ⟨ SB.Enumerations 3b, ... ⟩

    ⟨ SB.Imported types 4b, ... ⟩

    ⟨ SB.Signals 4d, ... ⟩

    ⟨ SB.Classes 5d, ... ⟩

    ⟨ SB.Associations 13b ⟩
end;
```

◇

Fragment referenced in 2.
Users: `Simple_Buttons` never used.

Signals correspond to events. UML has them declared at package (domain) level, though ColdFrame's implementation actually declares the corresponding event types in the specification of the class where they're used (in this case, Button; hence the need, in general, to specify the target class here (suppose there was more than one class in the domain that had to receive events?). Note the namespace separator `::`.

A class typically is an abstraction of something in the domain of interest. It represents the common properties and behaviour shared by all instances of the class.

An attribute holds a property of an object (either one per instance, for example the Accession Number of a Book in a Library, or per class, for example the next Accession Number to be used).

The purpose of operations is to implement the actual functionality of the domain.

An association is a relationship between two classes in the model (it is possible, though uncommon, to have a ıreflexive association between a class and itself, e.g. *Action is-a-consequence-of Action*).

2.1 Enumerations

This enumeration names the buttons. Only `B1` will be used.

⟨ *SB.Enumerations* 3b ⟩ ≡

```
enumeration Button_Name
   B1,
   B2
end;
```

◇

Fragment defined by 3b, 4a.
Fragment referenced in 3a.
Users: `Button_Name` in 6a.

This enumeration names the LEDs. Only `L1` will be used.

⟨ *SB.Enumerations* 4a ⟩ ≡

```
enumeration LED_Name
   L1,
   L2
end;
```

◇

Fragment defined by 3b, 4a.
Fragment referenced in 3a.
Users: `LED_Name` in 12b, 13a.

2.2 Imported Types

This imported type is used by the supporting Digital IO domain to report input (switch) state changes. The annotation `[imported]` includes a tagged value (tag `imported`, value `Digital_IO`).

⟨ *SB.Imported types* 4b ⟩ ≡

```
[imported (imported = Digital_IO)]
datatype Input_Signal_State;
```

◇

Fragment defined by 4bc.
Fragment referenced in 3a.
Users: `Input_Signal_State` in 8c.

This type is used by the supporting Digital IO domain to name outputs (LEDs).

⟨ *SB.Imported types* 4c ⟩ ≡

```
[imported (imported = Digital_IO)]
datatype Output_Signal;
```

◇

Fragment defined by 4bc.
Fragment referenced in 3a.
Users: `Output_Signal` in 13a.

2.3 Signals

This event indicates that the button 'pushed' period (after a short push) has expired.

⟨ *SB.Signals* 4d ⟩ ≡

```
signal Button::Lit_Timeout;
```

◇

Fragment defined by 4d, 5abc.
Fragment referenced in 3a.
Users: `Button::Lit_Timeout` in 11a.
Uses: `Button` 5d.

This event indicates that the button has been pushed.

⟨ *SB.Signals* 5a ⟩ ≡

```
signal Button::Push;
```

◇

Fragment defined by 4d, 5abc.
Fragment referenced in 3a.
Users: `Button::Push` in 5b, 10ab, 11ac.
Uses: `Button` 5d.

This event indicates that the button has been pushed long enough to make this a long push.

⟨ *SB.Signals* 5b ⟩ ≡

```
signal Button::Push_Timeout;
```

◇

Fragment defined by 4d, 5abc.
Fragment referenced in 3a.
Users: `Button::Push_Timeout` in 10b.
Uses: `Button` 5d, `Button::Push` 5a.

This event indicates that the button has been released.

⟨ *SB.Signals* 5c ⟩ ≡

```
signal Button::Release;
```

◇

Fragment defined by 4d, 5abc.
Fragment referenced in 3a.
Users: `Button::Release` in 10ab, 11c.
Uses: `Button` 5d.

2.4 Class Button

A Button controls a number of LEDs. When the Button is 'set', the LEDs related by A1 are lit.

Buttons respond to both 'short' and 'long' pushes.

After a long push, the button remains set until it's pushed again (long or short).

After a short push, the Button remains set for a period, which can be extended by a further short push or a long push.

⟨ *SB.Classes* 5d ⟩ ≡

```
class Button
    ⟨ SB.Button attributes 6a, ... ⟩

    ⟨ SB.Button operations 6e, ... ⟩

    ⟨ SB.Button state machine 9a ⟩
end;
◇
```

Fragment defined by 5d, 12a.
Fragment referenced in 3a.
Users: `Button` in 4d, 5abc, 9a, 10ab, 11ac, 13b.

2.4.1 Button attributes

This identifying attribute (the `id` modifier) is the name of the Button.

⟨ *SB.Button attributes* 6a ⟩ ≡

```
id attribute Name : Button_Name;
◇
```

Fragment defined by 6abcd.
Fragment referenced in 5d.
Uses: `Button_Name` 3b.

This attribute holds the time when the Button was pushed, so that the Lit timeout can run from this initial time rather than (e.g.) when the Button was released.

⟨ *SB.Button attributes* 6b ⟩ ≡

```
attribute Pushed_Time : Time;
◇
```

Fragment defined by 6abcd.
Fragment referenced in 5d.

This ColdFrame timer controls how long the Button needs to remain pushed before transition to the Held state.

⟨ *SB.Button attributes* 6c ⟩ ≡

```
attribute Lit_Timer : Timer;
◇
```

Fragment defined by 6abcd.
Fragment referenced in 5d.

This timer controls how long the Button needs to remain pushed before transition to the Held state.

⟨ *SB.Button attributes* 6d ⟩ ≡

```
attribute Pushed_Timer : Timer;
```

◇

Fragment defined by 6abcd.
Fragment referenced in 5d.

2.4.2 Button operations

The state of the button has changed; tell the controlled LEDs to reevaluate their own states (by checking whether any of the Buttons they are controlled by is set). Note the modifier `private`.

⟨ *SB.Button operations* 6e ⟩ ≡

```
private operation Changed();
```

◇

Fragment defined by 6e, 7abcd, 8abc.
Fragment referenced in 5d.

This operation stores the time at which the Button was pushed: the Lit timeout runs from this time, not the time of Button release.

This operation is short enough that we can include its code here, within the curly braces.

⟨ *SB.Button operations* 7a ⟩ ≡

```
private operation Note_Pushed_Time();
{
   This.Pushed_Time := ColdFrame.Project.Calendar.Clock;
}
```

◇

Fragment defined by 6e, 7abcd, 8abc.
Fragment referenced in 5d.

This operation sets the Pushed timeout, again including the code in the model. The indentation will be preserved (actually, relative to the first non-space character of the first line)

⟨ *SB.Button operations* 7b ⟩ ≡

```
private operation Set_Pushed_Timeout();
{
   ColdFrame.Project.Events.Set
     (The_Timer => This.Pushed_Timer,
      On => Events.Dispatcher,
      To_Fire => new Push_Timeout (This),
      After => 0.25);
}
```

◇

Fragment defined by 6e, 7abcd, 8abc.
Fragment referenced in 5d.

This operation clears the Pushed timeout.

⟨ *SB.Button operations* 7c ⟩ ≡

```
private operation Clear_Pushed_Timeout();
{
   ColdFrame.Project.Events.Unset
     (The_Timer => This.Pushed_Timer,
      On => Events.Dispatcher);
}
```
◇

Fragment defined by 6e, 7abcd, 8abc.
Fragment referenced in 5d.

This operation sets the Lit timeout. It's called on button release after a short push, but the time is relative to the time when the button was pushed.

⟨ *SB.Button operations* 7d ⟩ ≡

```
private operation Set_Lit_Timeout();
```
◇

Fragment defined by 6e, 7abcd, 8abc.
Fragment referenced in 5d.

This operation clears the Lit timeout.

⟨ *SB.Button operations* 8a ⟩ ≡

```
private operation Clear_Lit_Timeout();
{
   ColdFrame.Project.Events.Unset
     (The_Timer => This.Lit_Timer,
      On => Events.Dispatcher);
}
```
◇

Fragment defined by 6e, 7abcd, 8abc.
Fragment referenced in 5d.

This operation indicates whether the Button is set or not. It's set if it's in any of the states `Pushed`, `Held`, `Timed`, `Pushed_Again`.

Note, the code is emitted in the body of the Ada subprogram, so if any local variables are needed a `declare` block has to be used (in this particular case, a one-liner would actually have been possible).

⟨ *SB.Button operations* 8b ⟩ ≡

```
public operation Is_Set(): Boolean;
{
   declare
      Set_In_State : constant array (State_Machine_State_T) of Boolean
        := (Pushed | Held | Timed | Pushed_Again => True,
            others => False);
   begin
      return Set_In_State (This.State_Machine_State);
   end;
}
```

◇

Fragment defined by 6e, 7abcd, 8abc.
Fragment referenced in 5d.

This operation acts as receiver of state changes from Digital_IO, via Input Signal State Callback. The annotation `[callback]` triggers the necessary event generation. The modifier `static` isn't strictly necessary, since ColdFrame would automatically generate a class operation anyway, but avoids a warning.

Calls the instance Changed so the Button can take the appropriate action.

⟨ *SB.Button operations* 8c ⟩ ≡

```
[callback]
private static operation Receive_Change(S : Input_Signal_State);
```

◇

Fragment defined by 6e, 7abcd, 8abc.
Fragment referenced in 5d.
Uses: `Input_Signal_State` 4b.

2.4.3 Button state machine

This is a Moore model state machine; all the actions take place on entry to a state. See Figure 1 for the generated statechart.

ColdFrame also supports Mealy model state machines, where all the actions take place on transitions, as well as mixed machines.

⟨ *SB.Button state machine* 9a ⟩ ≡

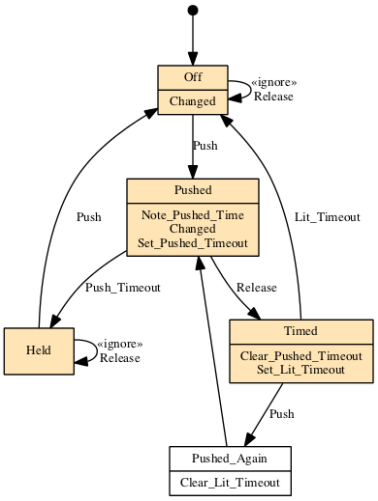
```
statemachine Button
   ⟨ SB.Button states 9b, ... ⟩
end;
```

◇

Fragment referenced in 5d.
Uses: `Button` 5d.

This is the inital state (indicated by the `init` modifier). It performs a completion transition to Off.

Figure 1: Generated Button statechart



⟨ *SB.Button states* 9b ⟩ ≡

```
initial state Initial
   transition to Off;
end;
```

◇

Fragment defined by 9b, 10ab, 11abc.
Fragment referenced in 9a.

In the state Off, the button is off, waiting for a Push. If this state was entered as a result of a Push in the Held state, there will be a corresponding Release, which is ignored (the annotation `[ignore]`).

On entry, `Changed` is called to tell the connected LEDs that they need to reconsider whether they should be lit.

⟨ *SB.Button states* 10a ⟩ ≡

```
state Off
   entry(Changed);
   transition on signal(Button::Push) to Pushed;
   [ignore] transition on signal(Button::Release) to Off;
end;
```

◇

Fragment defined by 9b, 10ab, 11abc.
Fragment referenced in 9a.
Uses: `Button` 5d, `Button::Push` 5a, `Button::Release` 5c.

In the state Pushed, the button is on, awaiting a Push_Timeout, which transitions to the Held state (a long push), or a Release (a short push), which transitions to the Timed state.

The entry actions are

1. Note_Pushed_Time: note when the button was pushed, so that if it's released before the coming Push_Timeout, this time can be used to determine how long the button remains 'pushed'.

2. Changed: tell the connected LEDs that they need to reconsider whether they should be lit.

3. Set_Pushed_Timeout: if this timeout occurs, this was a long push.

⟨ *SB.Button states* 10b ⟩ ≡

```
state Pushed
   entry(Note_Pushed_Time; Changed; Set_Pushed_Timeout);
   transition on signal(Button::Push_Timeout) to Held;
   transition on signal(Button::Release) to Timed;
end;
```

◇

Fragment defined by 9b, 10ab, 11abc.
Fragment referenced in 9a.
Uses: `Button` 5d, `Button::Push` 5a, `Button::Push_Timeout` 5b, `Button::Release` 5c.

In the state Timed, the button is on after a short push, awaiting a Lit_Timeout (which transitions to the Off state) or another Push (which transitions to Pushed_Again).

The entry actions are

1. Clear_Pushed_Timeout: The Pushed_Timeout that was started in the state Pushed is cancelled, because it's been overtaken by the short push that just occurred.

2. Set_Lit_Timeout: This determines how long the button remains 'pushed' for.

⟨ *SB.Button states* 11a ⟩ ≡

```
state Timed
   entry(Clear_Pushed_Timeout; Set_Lit_Timeout);
   transition on signal(Button::Push) to Pushed_Again;
   transition on signal(Button::Lit_Timeout) to Off;
end;
```

◇

Fragment defined by 9b, 10ab, 11abc.
Fragment referenced in 9a.
Uses: `Button` 5d, `Button::Lit_Timeout` 4d, `Button::Push` 5a.

In the state Pushed_Again, the button has been pushed during the timeout after a short push. Resets the timeout (in the entry action) and performs a completion transition to Pushed to start another check (this Push can be the start of another short push or a new long push).

⟨ *SB.Button states* 11b ⟩ ≡

```
state Pushed_Again
   entry(Clear_Lit_Timeout);
   transition to Pushed;
end;
```

◇

Fragment defined by 9b, 10ab, 11abc.
Fragment referenced in 9a.

In the state Held, the button is on, after a long push, awaiting another Push to transition to the Off state. The button is still pushed, so there will be a corresponding Release, which is ignored.

⟨ *SB.Button states* 11c ⟩ ≡

```
state Held
   transition on signal(Button::Push) to Off;
   [ignore] transition on signal(Button::Release) to Held;
end;
```

◇

Fragment defined by 9b, 10ab, 11abc.
Fragment referenced in 9a.
Uses: `Button` 5d, `Button::Push` 5a, `Button::Release` 5c.

Note that the state model could have been cast as a mixed Moore-Mealy machine, by writing the state Timed as

```
@d SB.Button states @{
state Timed
   entry(Clear_Pushed_Timeout; Set_Lit_Timeout);
   transition on signal(Button::Push) to Pushed
      do (Clear_Lit_Timeout);
   transition on signal(Button::Lit_Timeout) to Off;
end;
@}
```

which implements the `Clear_Lit_Timeout` action as the (only) effect of the transition signalled by the `Button::Pushed` event, and eliminates the need for the Pushed_Again state.

2.5 Class LED

An LED is lit when any of the Buttons it's controlled by is set.

⟨ *SB.Classes* 12a ⟩ ≡

```
class LED
   ⟨ SB.LED attributes 12b ⟩

   ⟨ SB.LED operations 12c, ... ⟩
end;
```

◇

Fragment defined by 5d, 12a.
Fragment referenced in 3a.
Users: `LED` in 13ab.

2.5.1 LED attributes

This attribute identifies the LED.

⟨ *SB.LED attributes* 12b ⟩ ≡

```
id attribute Name : LED_Name;
```

◇

Fragment referenced in 12a.
Uses: LED_Name 4a.

2.5.2 LED operations

This operation initialises the domain (this is indicated by the annotation `[init]`) by creating Button(s) and LED(s) as required, and associating them according to the required "circuit diagram".

⟨ *SB.LED operations* 12c ⟩ ≡

```
[init]
private static operation Initialize();
```

◇

Fragment defined by 12cd, 13a.
Fragment referenced in 12a.

This operation is called from a controlling Button which has changed to evaluate whether the LED should be lit (if any of the controlling Buttons is set) or not.

⟨ *SB.LED operations* 12d ⟩ ≡

```
public operation Changed();
```

◇

Fragment defined by 12cd, 13a.
Fragment referenced in 12a.

This operation maps the LED to the corresponding Digital_IO output pin.

⟨ *SB.LED operations* 13a ⟩ ≡

```
private operation Output_Signal_For_LED(): Output_Signal;
{
  --  This isn't going to be very extendable, but there's only one
  --  LED in this simple demo.
  return LED_Name'Pos (This.Name);
}
```

◇

Fragment defined by 12cd, 13a.
Fragment referenced in 12a.
Uses: LED 12a, LED_Name 4a, Output_Signal 4c.

2.6 Associations

This association relates each LED to the Button(s) it's controlled by.

Each Button controls one or more LEDs.

Each LED is controlled by one or more Buttons.

This is a many-to-many relationship, so ColdFrame requires that it be implemented as an Association Class, even though there are (as yet) no useful attributes for the Class part to contain.

⟨ *SB.Associations* 13b ⟩ ≡

```
association_class A1
   Button Controls LED[1,*];
   LED Is_Controlled_By Button[1,*];
end;
```

◇

Fragment referenced in 3a.
Users: `A1` never used.
Uses: `Button` 5d, `LED` 12a.

3 TextUML tokens

These are the tokens used (and, importantly, reserved) by TextUML. Those bolded correspond to stereotypes in ColdFrame.

abstract	enumeration	**null**	specializes
association	exception	on	state
association_class	false	operation	statemachine
attribute	**final**	out	static
class	**id**	package	terminate
component	in	primitive	to
datatype	initial	private	transition
do	inout	**protected**	true
end	interface	**public**	
entry	model	signal	

In most cases, there won't be a problem, but if you need to use one in an annotation (e.g. `[class]`, which at present is still needed in class signals and state machines – `static` should be allowed) you can either precede it with a backslash (`[\class]`) or capitalise it (`[Class]`).

Some of the ColdFrame stereotypes have hyphens, which isn't supported in TextUML because the name needs to be an identifier. Because of this, underscores in annotation names are translated to hyphens.

An example would be

`"test.tuml"` 14≡

```
model test;
   [domain_interface (name=test)]
   package test_it;
      [\protected] public datatype prot
         operation set(value : integer);
         [\entry] operation get(out value : integer);
         private attribute value : integer := 42;
```

```
          end;
       end;
    end.
    ◇
```

4 Syntax

Note, this syntax doesn't include the tokens; they are the UPPER CASE elements below. In most cases, the actual token is the lower-case version of the element here.

```
start : \
    model_comment annotations model_heading \
         namespace_contents END DOT

model_heading : MODEL qualified_identifier SEMICOLON

qualified_identifier \
    : identifier NAMESPACE_SEPARATOR qualified_identifier
    | identifier

namespace_contents \
    : top_level_element namespace_contents
    | top_level_element

sub_namespace \
    : package_heading \
      namespace_contents END SEMICOLON

package_heading : PACKAGE qualified_identifier SEMICOLON

top_level_element \
    : model_comment annotations top_level_element_choice

top_level_element_choice \
    : association_class_def
    | association_def
    | class_def
    | datatype_def
    | enumeration_def
    | exception_def
    | primitive_def
    | signal_def
    | sub_namespace

single_type_identifier : qualified_identifier

type_identifier \
```

```
    : single_type_identifier optional_multiplicity
    | function_signature optional_multiplicity

optional_multiplicity \
    : L_BRACKET multiplicity_spec R_BRACKET
    | empty

multiplicity_spec \
    : multiplicity_value COMMA multiplicity_value
    | multiplicity_value

association_def \
    : annotations ASSOCIATION identifier association_role_decl_list \
        END SEMICOLON

association_class_def \
    : annotations ASSOCIATION_CLASS identifier \
        association_role_decl_list feature_decl_list \
        END SEMICOLON
    | annotations ASSOCIATION_CLASS identifier \
        association_role_decl_list \
        END SEMICOLON

association_multiplicity \
    : L_BRACKET multiplicity_spec R_BRACKET

association_role_decl_list \
    : association_role_decl association_role_decl

association_role_decl \
    : model_comment annotations \
        identifier identifier identifier association_multiplicity SEMICOLON

class_def : class_header feature_decl_list END SEMICOLON

class_header \
    : class_modifiers class_type identifier class_specializes_section

class_modifiers \
    : class_modifier_list
    | empty

class_modifier_list \
    : class_modifier class_modifier_list
    | class_modifier

class_modifier \
```

```
    : visibility_modifier
    | ABSTRACT

class_specializes_section \
    : SPECIALIZES class_specializes_list
    | empty

class_specializes_list \
    : identifier COMMA class_specializes_list
    | identifier

class_type \
    : CLASS
    | INTERFACE
    | COMPONENT

feature_decl_list \
    : feature_decl feature_decl_list
    | feature_decl

feature_decl \
    : model_comment annotations feature_modifiers feature_type

feature_modifiers \
    : feature_modifier_list
    | empty

feature_modifier_list \
    : feature_modifier feature_modifier_list
    | feature_modifier

feature_modifier \
    : visibility_modifier
    | STATIC
    | ABSTRACT
    | ID

visibility_modifier \
    : PUBLIC
    | PRIVATE
    | PACKAGE
    | PROTECTED

feature_type \
    : state_machine_decl
    | operation_decl
    | attribute_decl
```

```
state_machine_decl \
    : STATEMACHINE identifier state_decls END SEMICOLON
    | STATEMACHINE state_decls END SEMICOLON

state_decls \
    : state_decl state_decls
    | state_decl

state_decl \
    : model_comment state_modifier STATE identifier state_behaviours \
        transition_decls END SEMICOLON
    | model_comment STATE identifier state_behaviours \
      transition_decls END SEMICOLON

state_modifier \
    : INITIAL
    | TERMINATE
    | FINAL

state_behaviours \
    : state_behaviour_list
    | empty

state_behaviour_list \
    : state_behaviour state_behaviour_list
    | state_behaviour

state_behaviour : ENTRY state_behaviour_definition SEMICOLON

state_behaviour_definition : simple_statement_block

transition_decls \
    : transition_decl_list
    | empty

transition_decl_list \
    : transition_decl transition_decl_list
    | transition_decl

transition_decl \
    : model_comment annotations TRANSITION ON SIGNAL \
        L_PAREN qualified_identifier R_PAREN \
        TO identifier transition_effect_opt SEMICOLON
    | model_comment annotations TRANSITION TO identifier \
        transition_effect_opt SEMICOLON
```

```
transition_effect_opt \
    : DO simple_statement_block
    | empty

simple_statement_block \
    : L_PAREN statement_list R_PAREN
    | identifier

statement_list \
    : identifier SEMICOLON statement_list
    | identifier

operation_body : OPERATION_BODY

operation_decl \
    : operation_header SEMICOLON operation_body
    | operation_header SEMICOLON

operation_header : OPERATION identifier signature

attribute_decl \
    : ATTRIBUTE identifier COLON type_identifier \
        initialization_expression_opt SEMICOLON

initialization_expression_opt \
    : initialization_expression
    | empty

initialization_expression : ASSIGNOP simple_initialization

simple_initialization : literal_or_identifier

function_signature : L_CURLY_BRACKET simple_signature R_CURLY_BRACKET

signature : L_PAREN param_decl_list R_PAREN optional_return_type

simple_signature \
    : L_PAREN simple_param_decl_list R_PAREN simple_optional_return_type
    | L_PAREN simple_param_decl_list R_PAREN

optional_return_type \
    : annotations simple_optional_return_type
    | empty

simple_optional_return_type : COLON type_identifier

param_decl_list \
```

```
    : param_decl COMMA param_decl_list
    | param_decl
    | empty

simple_param_decl_list \
    : simple_param_decl COMMA simple_param_decl_list
    | simple_param_decl
    | empty

param_decl : annotations parameter_modifiers simple_param_decl

simple_param_decl \
    : optional_parameter_name COLON type_identifier \
        initialization_expression_opt

optional_parameter_name \
    : identifier
    | empty

parameter_modifiers \
    : parameter_modifier parameter_modifiers
    | empty

parameter_modifier \
    : IN
    | OUT
    | INOUT

annotations \
    : L_BRACKET annotation_list R_BRACKET
    | empty

annotation_list \
    : annotation COMMA annotation_list
    | annotation

annotation \
    : qualified_identifier annotation_value_specs
    | qualified_identifier

annotation_value_specs \
    : L_PAREN annotation_value_spec_list R_PAREN

annotation_value_spec_list \
    : annotation_value_spec COMMA annotation_value_spec_list
    | annotation_value_spec
```

```
annotation_value_spec : identifier EQUALS annotation_value

annotation_value \
    : literal
    | qualified_identifier

datatype_def \
    : datatype_header feature_decl_list END SEMICOLON
    | datatype_header SEMICOLON

datatype_header : class_modifiers DATATYPE identifier

enumeration_def \
    : visibility_modifier ENUMERATION identifier \
        enumeration_literal_decl_list END SEMICOLON
    | ENUMERATION identifier \
        enumeration_literal_decl_list END SEMICOLON

enumeration_literal_decl_list \
    : enumeration_literal_decl enumeration_literal_decl_list_tail

enumeration_literal_decl : model_comment identifier

enumeration_literal_decl_list_tail \
    : COMMA enumeration_literal_decl_list
    | empty

exception_def \
    : visibility_modifier EXCEPTION identifier SEMICOLON
    | EXCEPTION identifier SEMICOLON

signal_def : signal_decl

signal_decl \
    : SIGNAL qualified_identifier signal_attributes END SEMICOLON
    | SIGNAL qualified_identifier SEMICOLON

signal_attributes \
    : signal_attribute_decl signal_attributes
    | signal_attribute_decl

signal_attribute_decl \
    : ATTRIBUTE identifier COLON type_identifier SEMICOLON

primitive_def \
    : visibility_modifier PRIMITIVE identifier SEMICOLON
    | PRIMITIVE identifier SEMICOLON
```

```
model_comment \
    : MODEL_COMMENT
    | empty

identifier : IDENTIFIER

literal \
    : boolean
    | number
    | STRING
    | NULL

literal_or_identifier \
    : literal
    | identifier

boolean \
    : TRUE
    | FALSE

number \
    : INTEGER
    | REAL

multiplicity_value \
    : INTEGER
    | MULT
```

5 Files

"test.tuml" Defined by 14.
"textuml.tuml" Defined by 2.

6 Macros

⟨ SB.Associations 13b ⟩ Referenced in 3a.
⟨ SB.Button attributes 6abcd ⟩ Referenced in 5d.
⟨ SB.Button operations 6e, 7abcd, 8abc ⟩ Referenced in 5d.
⟨ SB.Button state machine 9a ⟩ Referenced in 5d.
⟨ SB.Button states 9b, 10ab, 11abc ⟩ Referenced in 9a.
⟨ SB.Classes 5d, 12a ⟩ Referenced in 3a.
⟨ SB.Enumerations 3b, 4a ⟩ Referenced in 3a.
⟨ SB.Imported types 4bc ⟩ Referenced in 3a.
⟨ SB.LED attributes 12b ⟩ Referenced in 12a.
⟨ SB.LED operations 12cd, 13a ⟩ Referenced in 12a.
⟨ SB.Signals 4d, 5abc ⟩ Referenced in 3a.
⟨ The domains 3a ⟩ Referenced in 2.

7 Definitions

A1: defined in 13b, never used.
Button: defined in 5d, used in 4d, 5abc, 9a, 10ab, 11ac, 13b.
Button::Lit_Timeout: defined in 4d, used in 11a.
Button::Push: defined in 5a, used in 5b, 10ab, 11ac.
Button::Push_Timeout: defined in 5b, used in 10b.
Button::Release: defined in 5c, used in 10ab, 11c.
Button_Name: defined in 3b, used in 6a.
Input_Signal_State: defined in 4b, used in 8c.
LED: defined in 12a, used in 13ab.
LED_Name: defined in 4a, used in 12b, 13a.
Output_Signal: defined in 4c, used in 13a.
Simple_Buttons: defined in 3a, never used.