Object Orientation

NEW IN 1.1

Related Publications

Towards Object-Oriented Modeling in SCCharts. Alexander Schulz-Rosengarten and Steven Smyth and Michael Mendler. In Proc. Forum on Specification and Design Languages (FDL '19), Southampton, UK, 2019.

Inheritance

SCCharts support inheritance, similar to the concept of referenced SCCharts (macro expansion).

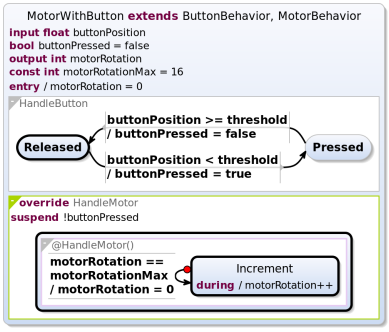
Each root state can extend multiple base states (since the name super states is already used) by listing them after the **extends** keyword.

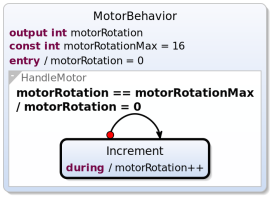
Such a state will inherit all declarations, action and regions of the base states. If conflicts arise or a state has a cyclic inheritance hierarchy, a warning will be displayed. If a state is contained multiple times in the inheritance hierarchy its content will inherited only once.

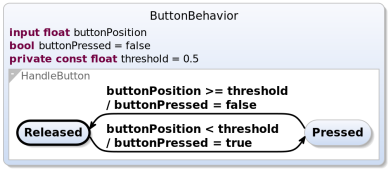
Declarations can have the **private** keyword to prevent extending SCCharts from accessing these variables.

Root-level regions of base states can be overridden with the **override** keyword and consequently replace the behavior by the new definition. Anonymous regions (defined without an ID) cannot be overridden.

Regions can also be references similar to states by using the **is** keyword. If the reference should refer to the implementation in the base state, the **super** keyword must be used.







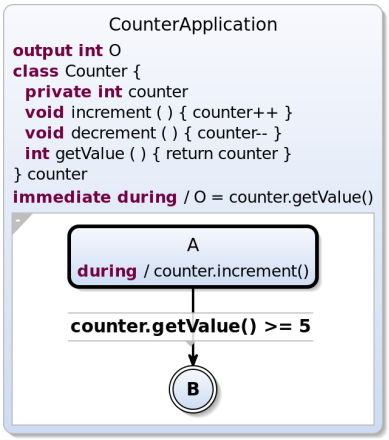
```
scchart MotorWithButton
  extends ButtonBehavior,
MotorBehavior {

  override region HandleMotor {
    suspend if !buttonPressed

    region is super.HandleMotor
  }
}

scchart MotorBehavior {
  output int motorRotation

  const int motorRotationMax =
16
  entry do motorRotation = 0

  region HandleMotor {
    initial state Increment {
      during do motorRotation++
    } if motorRotation ==
motorRotationMax
      do motorRotation = 0
      abort to Increment
  }
}

scchart ButtonBehavior {
  input float buttonPosition

  bool buttonPressed = false
  private const float threshold
= 0.5

  region HandleButton {
    initial state Released
      if buttonPosition <
threshold
      do buttonPressed = true
      go to Pressed

    state Pressed
      if buttonPosition >=
threshold
      do buttonPressed = false
      go to Released
  }
}
```

Classes

Class declarations allow to define hierarchical data structures. They may contain variable and method declarations as members. This includes inner class declarations.

In SCCharts, variables of a class type are statically instantiated. Hence, read/write access is only permitted on members.

Note there are also **struct** declarations that are a subset of class declarations since they prohibit the optional declaration of methods.



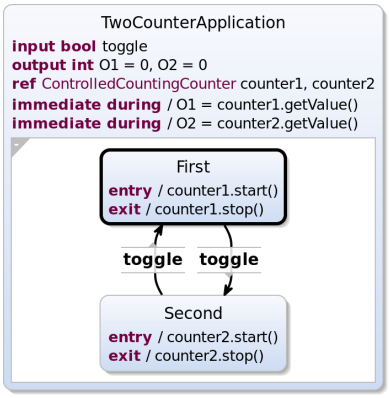
```
scchart CounterApplication {

  output int O

  class Counter {
    private int counter

    void increment() {
      counter++
    }
    void decrement() {
      counter--
    }
    int getValue() {
      return counter
    }
  } counter

  immediate during do O =
counter.getValue()

  initial state A {
    during do counter.
increment()
  }
  if counter.getValue() >= 5
  go to B

  final state B

}
```

Class Modeling

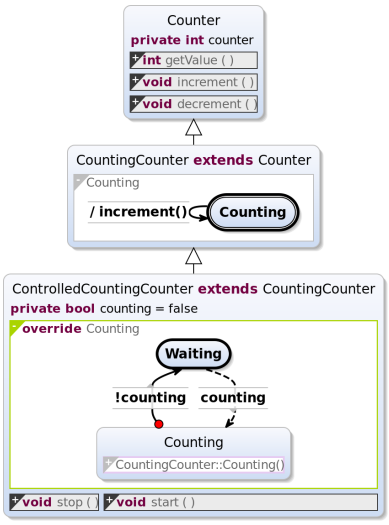
As an alternative to class declarations, classes can also be modeled using SCCharts. Every SCChart, that has no input or output variables can be used as a class definition. The most important advantage is that you can use inheritance in your class design. It is also possible to declare variables with a **protected** visibility. Furthermore, SCCharts now can define methods in addition to variables and regions, that might be helpful even if the SCChart is not used as an class definition.

Use **ref** declarations, as in SCCharts' Dataflow, to declare SCCharts-based classes. Same as class declarations these classes are statically instantiated for each variable. All regions in the SCCharts class instances will immediately start when the scope of the variable is entered.

```
import controlled-counting-
counter

scchart TwoCounterApplication {
  input bool toggle
  output int O1 = 0, O2 = 0

  ref ControlledCountingCounter
counter1, counter2

  immediate during do O1 =
counter1.getValue()
  immediate during do O2 =
counter2.getValue()

  region {
    initial state First {
      entry do counter1.start()
      exit do counter1.stop()
    } if toggle
      go to Second

    state Second {
      entry do counter2.start()
      exit do counter2.stop()
    } if toggle
      go to First
  }
}
```



```
scchart
ControlledCountingCounter
extends CountingCounter {
  private bool counting = false

  public method start() {
    counting = true
  }
  public method stop() {
    counting = false
  }

  override region Counting {
    initial state Waiting
      immediate if counting go
to Counting

    state Counting {
      region Counting is super.
Counting
    } if !counting abort to
Waiting
  }
}

scchart CountingCounter extends
Counter {
  region Counting {
    initial final state Counting
      do increment()
      go to Counting
  }
}

scchart Counter {
  private int counter

  method increment() {
counter++ }
  method decrement() {
counter-- }
  method int getValue() {
return counter }
}
```

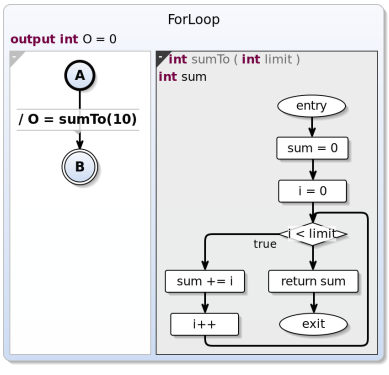


Methods

Methods can be declared in classes, states and regions. They can be invoked in expressions, effects on transitions, and entry /exit/during actions w.r.t. scoping and visibility. Methods can take parameters and return a value. They can also access any variable in their scope (enclosing states/classes).

Method bodies contain imperative immediate code sections consisting of a restricted set of SCL (assignments, method calls, labels, gotos, return statements, if/else statements, and for or while loops).

```
scchart ForLoop {
  output int O = 0

  method int sumTo(int limit) {
    int sum = 0
    for (int i = 0; i < limit;
i++) {
      sum += i
    }
    return sum
  }

  region {
    initial state A
      do O = sumTo(10)
      go to B

    final state B
  }
}
```



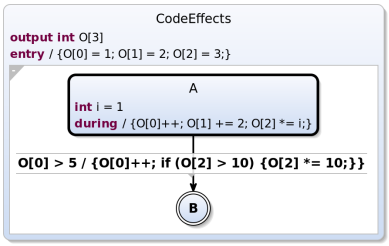
To handle method calls the compiler usually inlines the body. Hence, each method call acts as a macro expansion. However, since the netlist-based approach does not support loops it uses a different strategy. If the method body contains loops or is rather long it will not be inlined but kept as a function even in the generated code. As a consequence, this approach cannot handle method calls in a concurrent context that require interleaving because without inlining the method calla are considered atomic. Hence, some programs might be rejected by the compiler. You can annotate method declarations with **@inline** to advice the compiler to inline this method.
The priority-based approach will always inline all method calls.
You can influence the default handling of methods in a compilation system by setting the respective compiler properties (see Method Processor)

Code Effects

Code effects are a shortcut notation for anonymous parameter-less method calls in effects of transitions and entry/exit/during actions.

Code effects follow the same rules as method bodies.

```
scchart CodeEffects {
  output int O[3]

  entry do {
    O[0] = 1;
    O[1] = 2;
    O[2] = 3;
  }

  initial state A {
    int i = 1
    during do {
      O[0]++;
      O[1] += 2;
      O[2] *= i;
    }
  }
  if O[0] > 5
  do {
    O[0]++;
    if (O[2] > 10) {
      O[2] *= 10;
    }
  } go to B

  final state B
}
```



Generics

COMING SOON

Host Classes

Class declarations also allow more advanced object oriented host code integration. Using the **host** keyword, the class will be treated as a host code type. The declaration allows to mimic the objects API with fields and methods by defining members. These members do not affect the code generation since the host class will be used directly from the host languages. However, in the SCChart itself this allows proper oo access to the host object.

Host classes can be augmented by Scheduling Directives that will affect the ordering of method calls in the SCChart.

#resource includes external files in the compilation and simulation of SCCharts.

#hostcode allows to insert host code above the generated code.

Note that the this host class integration is not limited to Java, since host structs can be used for C. Additionally, the generated C code is usually c++ compatible. A struct in C used as a host variable is expected to be defined as typedef not a named struct.

- **counter.c**
- **counter.h**
- **counter-application-c.sctx**

```
#resource "Counter.java"

scchart HostCounterApplication {

  input bool CountUp
  output int O

  host class Counter {
    void increment()
    void decrement()
    int getValue()
  } counter

  region {
    initial state A ""
      if CountUp do counter.
increment();
                 O = counter.
getValue() go to A
      if !CountUp do counter.
decrement();
                 O = counter.
getValue() go to A
  }
}
```

```
class Counter {
    private int value = 0;

    public void increment() {
        value++;
    }
    public void decrement() {
        value--;
    }
    public int getValue() {
        return value;
    }
}
```

```
#hostcode "import java.util.
List;"
#hostcode "import java.util.
LinkedList;"

scchart UsingJavaList {

  output string info = "[]"
  int size = 99

  host class "List<Integer>" {
    private schedule
{commuting, commuting} order
    bool add(int v) schedule
order 0
    int size() schedule order 1
    string toString() schedule
order 1
  } list = `new LinkedList()`

  during do list.add(size);
list.add(size + 1)
  during do size = list.size()
  during do info = list.
toString()
}
```

Enums

UPCOMING 1.3 (NIGHTLY)

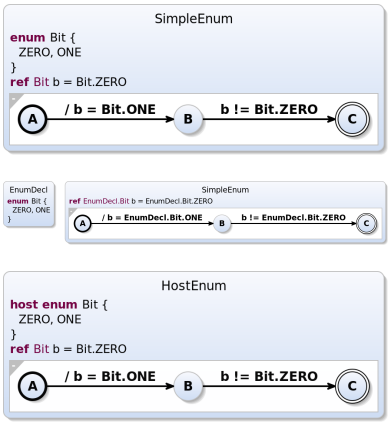
Even if enumerations are not a part of object orientation, they are added to SCCharts using the same object principle and notation.

Syntax and usage is inspired by Java rather than C, especially w.r.t. scoping.

An **enum** declaration declares the enumeration itself not a variable of this type (c. f. static instantiation of classes). A **ref** declaration then creates a variables that can hold enum values.

To access an enum type declared in a different SCChart the **scchart** accessor can be used to access the declaration (see example HostEnum)

```
scchart SimpleEnum {

  enum Bit { ZERO, ONE }

  ref Bit b = Bit.ZERO

  initial state A
   do b = Bit.ONE go to B

  state B
   if b != Bit.ZERO go to C

  final state C
}
```



The transformation will completely remove enumerations and will replace them by integer values. Hence, enumeration can be considered and alternative to named constants.

An enumeration can be declared as **host enum** requiring it to be present in the host language.

```
scchart SimpleEnum {

  ref EnumDecl.Bit b = scchart
(EnumDecl).Bit.ZERO

  initial state A
   do b = scchart(EnumDecl).Bit.
ONE go to B

  state B
   if b != scchart(EnumDecl).
Bit.ZERO go to C

  final state C
}

scchart EnumDecl {
  enum Bit { ZERO, ONE }
}
```

```
#hostcode-java-inner "enum Bit
{ZERO,ONE}"
#hostcode-c-header "enum Bit
{ZERO,ONE};"

scchart HostEnum {
  host enum Bit { ZERO, ONE }

  ref Bit b = Bit.ZERO

  initial state A
   do b = Bit.ONE go to B

  state B
   if b != Bit.ZERO go to C

  final state C

}
```