Grant Agreement: 287829

Comprehensive Modelling for Advanced Systems of Systems

COMPASS

Fourth Release of the COMPASS Tool
SysML to CML Translation

Deliverable Number: D31.4d

Version: 1.0

Date: September 2014

Public Document

http://www.compass-research.eu

Contributors:

Alvaro Miyazawa, York
Peter Gorm Larsen, Aarhus

Editors:

Joey W. Coleman, Aarhus

Reviewers:

Uwe Schulze, Bremen
Richard Payne, Newcastle
Adrian Larkham, Atego

Document History

Ver	Date	Author	Description
0.1	15-07-2014	JWC	Initial document setup from internal S2C report
0.2	18-07-2014	AM	Add overview of approach; expand explanation in 1st example
0.3	21-07-2014	JWC	Editing
0.4	23-07-2014	PGL	Adding background for the mitigation action that this results from
0.5	23-07-2014	JWC	Editing; ready for internal review.
0.6	21-07-2014	AHM	Addressed reviewers comments.
1.0	22-08-2014	JWC	Ready for delivery

Contents

1 Introduction **5**

2 Technical considerations **5**
2.1 Simplifying assumptions 5
2.2 Translation approach 6

3 Examples **8**
3.1 Basic example 1 8
3.2 Streaming example 10
3.3 Dwarf example 15
3.4 Hybrid SUV 17

4 Conclusions **24**

1 Introduction

The semantics of SysML models specified in Deliverable D22.4 [MCI+13] is a formal denotational semantics. It is very general, compositional and comprehensive, and, for this reason, requires a number of specialised CML modelling patterns that support the integration of all the features within a CML semantics, and, as a consequence, increases the size and complexity of the resulting semantic models. For instance, since CML does not provide direct support for asynchronous communication, and SysML requires it, an asynchronous communication layer must be modelled and added to the semantics of SysML models.

When the first stable versions implementing the semantics rules in converting SysML models from Artisan studio to CML became available it became apparent that the sheer size of the generated CML models was an issue. In particular the industrial users from Theme 4 of the COMPASS project had difficulties in recognising the components of their SysML model in the generated CML model. In order to mitigate the risk that the translation between SysML and CML would be considered problematic, a small task force was organised to attempt to investigate an alternative translation model. This translation used a synchronous semantic model for translating SysML state machines to CML; this clearly uses a different semantic model than that in the OMG SysML standard. The resulting tool feature from this mitigation action is called S2C-lite and it is built into the Symphony IDE. It is able to translate XMI representations of restricted SysML models into CML models.

Section 2 discusses the assumptions taken in order to simplify the generated models and the approach to the translation of SysML models. Section 3 presents the results of the application of the translation tool to a number of examples, and Section 4 summarises the results achieved in this work.

2 Technical considerations

2.1 Simplifying assumptions

S2C-lite makes it possible, under certain simplifying assumptions, to obtain simpler semantic models that are closer to the specifications that would otherwise be written in CML. In this work, we explore some stronger restrictions over SysML models and their use in order to obtain simpler and more readable CML models. This simplified semantics is implemented in the S2C-lite plug-in that is accessible in the Symphony IDE.

The simplifying assumptions made in this work are as follows:

- Blocks are only considered in terms of their attributes and data operations. That is, interactions between blocks through communication (e.g., signal events) are ignored;
- Primitive types are assumed to correspond to CML types;
- Only primitive types, data types, value types, enumerations, and blocks are used as types of properties and parameters;

- State machines do not contain parallel constructs. This restriction excludes parallel regions, as well as do-activities, which under certain circumstances can lead to parallel execution;
- State machines do not contain history, fork or join pseudo-states. The first is excluded due to reduced coverage, and the others are not necessary since parallel regions are not allowed;
- Transitions can only be triggered by signal events, and these events are modelled as *synchronous* communications. This is the main departure from the official SysML semantics, but it is, in fact, often adopted in industrial practice. Some of the consequences of this deviation are further discussed in this report;
- The action language for SysML actions is the subset of CML that includes data operations and channel communications encoding the sending of signals;
- All elements of the model must have a unique name;
- Each state machine and composite state should have at most one initial state;
- Each state machine must be inside a block and only use those attributes and operations that are in scope;
- No other aspects of SysML models are considered in the implemented semantics.

As described above, a number of restrictions over SysML models have been imposed to allow the generation of simple, executable and readable models. The main focus of this semantics is the behaviour of single state machines. That is, the interactions between two or more state machines is not considered, as this feature increases the complexity of the models making them less readable. Additionally, the interaction between multiple blocks and state machines is considered in the original semantics in D22.4.

The S2C-lite tool assigns a synchronous semantics to state machine events, which, as a consequence, offers only those events that actively affect the state machine. This is in contrast with the original semantics of SysML state machines, in which a state machine does not refuse an event (it can defer it, however), but discards any events that do not trigger an internal reaction.

The most important restrictions is the absence of parallel states. This is necessary due to the fact that CML does not allow shared variables, and the models of parallel state machines requires the modelling of shared variables in terms of communication, which leads to a more complicated model.

2.2 Translation approach

In general, each state produces two CML actions that encode (1) its activation (entry action), its substates' activations, its deactivation and its transitions, and (2) its deactivation and the deactivation of its substates. The reason for the separation of the deactivation action is that it must consider all possible configurations of substates, and increase the size of the action that specifies the behaviour of the state, masking the actual behaviour being encoded.

The transitions of a state are modelled as actions that are offered in an external choice (if more than one transition is available).

- Completion transitions (those without triggers) are offered before non-completion transitions (those with triggers) and are modelled as an external choice of guarded actions (`[guard] & Action`). The guards form a cover (that is, the disjunction of all the guards is true).
- Non-completion transitions are modelled as an external choice of prefixing actions (`comm -> Action`) that are only offered if none of the completion transitions can be executed.

Interlevel transitions (that is, transitions whose source and target states are not siblings) are modelled by first identifying the least common ancestor of the transitions, and, when exiting the source state and entering the target state, executing the exit and entry actions in the appropriate order, including any parents that are substates of the least common ancestor.

3 Examples

This section presents examples that have been used to validate the S2C lite SysML to CML translation tool and its output. It is worth mentioning that all these examples use CML as an the language for conditions and actions.

Since the focus of this semantics is the state machines, in the following sections we only present the SysML diagrams of state machines. In particular, operations defined in a SysML models via CML actions are stored in the model bu do not appear in any of the diagrams.

3.1 Basic example 1

The example in Figure 1 was constructed to be a starting point for the development of the tool. It contains the minimal features of SysML state machines deemed necessary for modelling interesting behaviours.

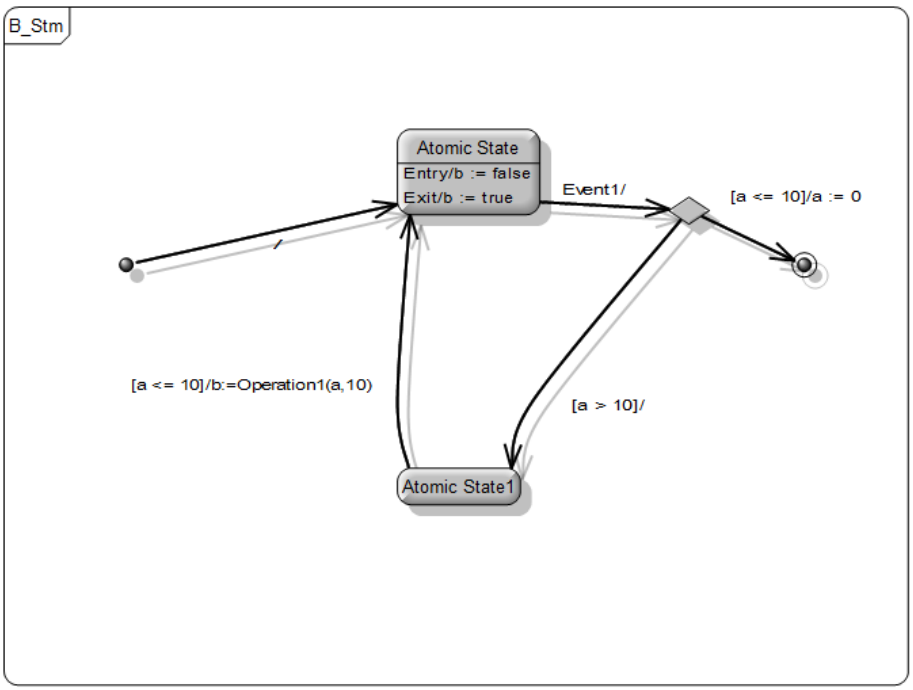

Figure 1: Basic state machine

This example contains only simple states, entry and exit actions, junctions, transitions, triggers, guards and transition actions. The block `B` (not shown in the figures) contains three attributes `a`, `b` and `c`. Whilst the first two are used by the state machine, the latter is not, but appears in the definition of the class B and process B_Stm as it is part of the model. The result of applying the translation tool to the associated SysML model is shown below.

```
channels
  Signal1 : int

class B = begin
state
  a : int
  b : bool
  c : seq of char
end

process B_Stm = begin
state
  a : int
  b : bool
  c : seq of char

operations
  Operation1 : int * int ==> bool
  Operation1(x, y) == return x>y

actions
  act_exit_Final = Skip

  act_Final = Stop

  act_exit_Atomic_State = b := true

  act_Atomic_State = b := false ; (
    (Signal1?x -> act_exit_Atomic_State;act_Junction)
  )

  act_exit_Atomic_State1 = Skip

  act_Atomic_State1 = (
    ([a <= 10]&(b:=Operation1(a,10) ; act_Atomic_State))
  )

  act_exit_Initial = Skip

  act_Initial = (
    (act_Atomic_State)
  )

  act_exit_Junction = Skip

  act_Junction = (
    ([a > 10]&(act_Atomic_State1))
    []
    ([a <= 10]&(a := 0 ; act_Final))
  )

  @ act_Initial
end
```

For the purpose of checking the consistency in the use of types, each block in the model yields a class definition. In this example, since there is a single block, only the class `B` is generated. Furthermore, a CML process that models the state machine is generated `B_Stm`. It includes the contents of the associated class because it potentially acts upon the block's properties and operations.

Simple states, such as `Atomic State`, are translated into two CML actions, one modelling the behaviour of the state itself, and the other modelling the exiting behaviour of the process. This is necessary due to dependencies on the exit order when considering composite states. In the example in Figure 1 the actions associated with the state `Atomic State` are `act_Atomic_State` and `act_exit_Atomic_State`. The first action assigns **`false`** to `b` and waits for a communication on the channel `Signal1` (that corresponds to the event `Event1`), calls its exit action `act_exit_Atomic_State`, and proceeds to the choice state `Junction` by calling its associated action `act_Junction`.

Final states are modelled as the **`Stop`** actions, and multiple transitions are modelled by an external choice.

3.2 Streaming example

The example in Figure 2 is based on a model used to test a streaming system in RT-Tester. It required the translation of guards and actions into CML expressions and statements. The generated model of this example uncovered an idiosyncrasy in the use of enumeration types, which cannot be replicated in our translation tool without embedding certain assumptions. In this case, there is an association between enumerated types and natural numbers, where enumeration literals were treated as numbers in the conditions of the state machine.

This use of enumeration is not compatible with our translation of enumeration types as the union of quote types, and, in order to produce correct CML model, it was necessary to add constants for the enumeration literals with the appropriate integer value, and add the attributes that are spread across the different blocks to the process. Notice that this use of enumerations is particular to this model and is not imposed by the modelling tool.

Whilst this example can be simulated, the usefulness of the generated model is unclear due to the lack of events in the transitions. This is due to the fact that in CML, it is only possible to interact with a process through communications, and the lack of events in the transitions translates into a lack of communications in the generated model. This can be overcome by introducing signals in the transitions, which would allow the inspection of the model at particular states during simulation.

The generated model is shown below.

```
channels
  Do

types
  float ::
  signed_char ::
  signed_int ::
  timer ::
  unsigned_char ::
  unsigned_int ::
  void ::

class SYSTEM = begin
state
  SystemUnderTest : SystemUnderTest
  TestEnvironment : TestEnvironment
```

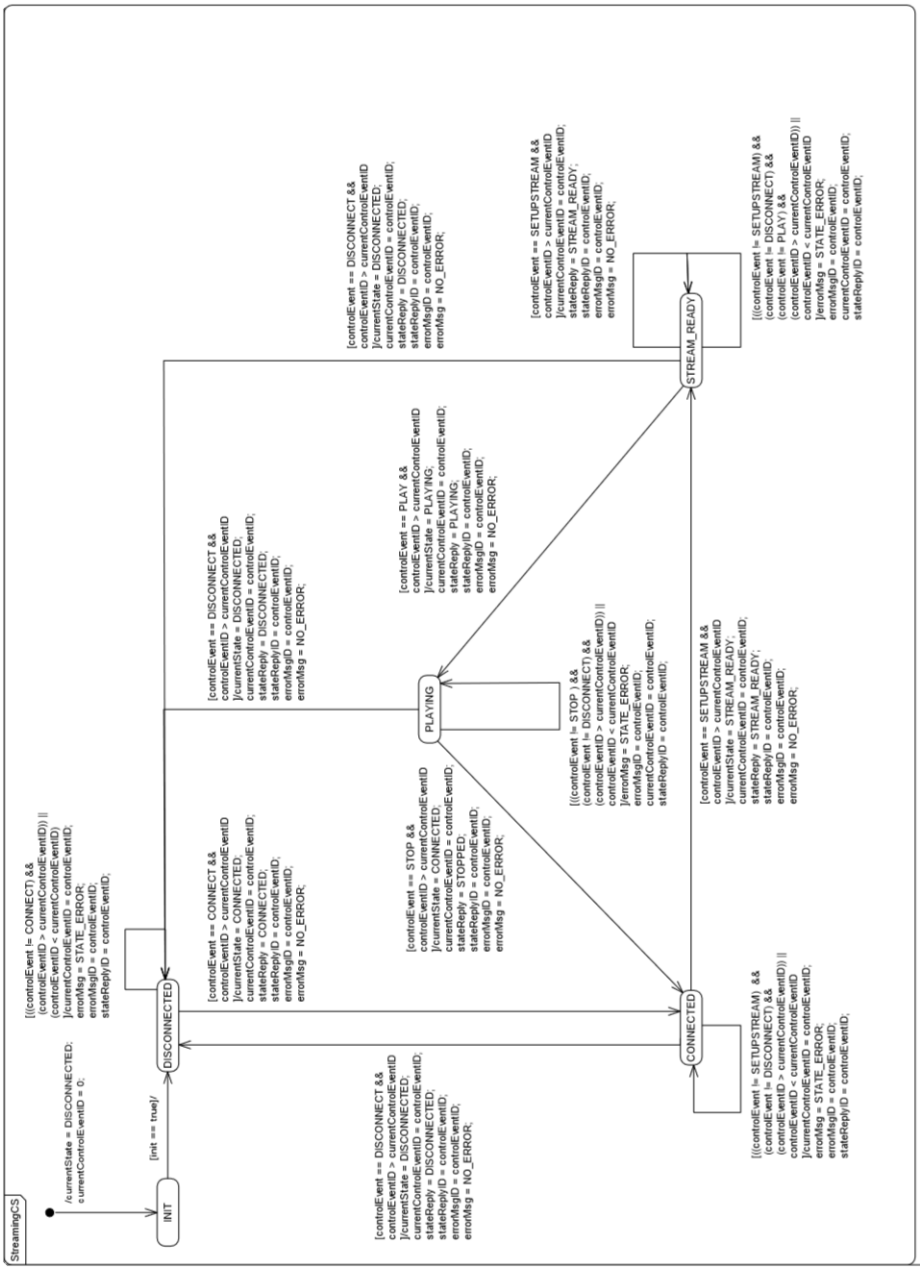

Figure 2: Streaming example

```
  replyEvent : StreamingReplyEvent
  errorEvent : StreamingErrorEvent
  controlEvent : StreamingControlEvent
end

class SystemUnderTest = begin
state
  currentControlEventID : int
  currentStateEventReplyID : int
  currentState : int
  stateMachine : StreamingCS
end

class StreamingCS = begin
end

class TestEnvironment = begin
end

class StreamingReplyEvent = begin
state
  stateReplyID : int
  stateReply : int
end

class StreamingErrorEvent = begin
state
  errorMsgID : int
  errorMsg : int
end

class StreamingControlEvent = begin
state
  controlEventID : int
  controlEvent : int
  init : bool
end

process StreamingCS = begin
state
  currentControlEventID : int
  currentStateEventReplyID : int
  currentState : int
  stateReplyID : int
  stateReply : int
  errorMsgID : int
  errorMsg : int
  controlEventID : int
  controlEvent : int
  init : bool

actions
act_exit_CONNECTED = Skip
act_CONNECTED = (
  (
    [((controlEvent <> SETUPSTREAM) and
     (controlEvent <> DISCONNECT) and
     (controlEventID > currentControlEventID)) or
      controlEventID < currentControlEventID
    ]&(currentControlEventID:= controlEventID;
      errorMsg:= STATE_ERROR;
      errorMsgID:= controlEventID;
```

```
      stateReplyID:= controlEventID;  act_CONNECTED)
  )
  []
  (
    [controlEvent = DISCONNECT and
    controlEventID > currentControlEventID
    ]&(currentState:= DISCONNECTED;
      currentControlEventID:= controlEventID;
      stateReply:= DISCONNECTED;
      stateReplyID:= controlEventID;
      errorMsgID:= controlEventID;
      errorMsg:= NO_ERROR;  act_DISCONNECTED)
  )
  []
  (
    [controlEvent = SETUPSTREAM and
    controlEventID > currentControlEventID
    ]&(currentState:= STREAM_READY;
      currentControlEventID:= controlEventID;
      stateReply:= STREAM_READY;
      stateReplyID:= controlEventID;
      errorMsgID:= controlEventID;
      errorMsg:= NO_ERROR;  act_STREAM_READY)
  )
)

act_exit_DISCONNECTED = Skip

act_DISCONNECTED = (
  (
    [controlEvent = CONNECT and
    controlEventID > currentControlEventID
    ]&(currentState:= CONNECTED;
      currentControlEventID:= controlEventID;
      stateReply:= CONNECTED;
      stateReplyID:= controlEventID;
      errorMsgID:= controlEventID;
      errorMsg:= NO_ERROR;  act_CONNECTED)
  )
  []
  (
    [((controlEvent <> CONNECT) and
     (controlEventID > currentControlEventID)) or
     (controlEventID < currentControlEventID)
    ]&(currentControlEventID:= controlEventID;
      errorMsg:= STATE_ERROR;
      errorMsgID:= controlEventID;
      stateReplyID:= controlEventID;  act_DISCONNECTED)
  )
)

act_exit_INIT = Skip
act_INIT = (
  ([init = true]&(act_DISCONNECTED))
)

act_exit_PLAYING = Skip
act_PLAYING = (
  (
    [controlEvent = STOP and
    controlEventID > currentControlEventID
    ]&(currentState:= CONNECTED;
```

```
      currentControlEventID:= controlEventID;
      stateReply:= STOPPED;
      stateReplyID:= controlEventID;
      errorMsgID:= controlEventID;
      errorMsg:= NO_ERROR;  act_CONNECTED)
  )
  []
  (
    [controlEvent = DISCONNECT and
    controlEventID > currentControlEventID
    ]&(currentState:= DISCONNECTED;
      currentControlEventID:= controlEventID;
      stateReply:= DISCONNECTED;
      stateReplyID:= controlEventID;
      errorMsgID:= controlEventID;
      errorMsg:= NO_ERROR;  act_DISCONNECTED)
  )
  []
  (
    [((controlEvent <> STOP ) and
    (controlEvent <> DISCONNECT) and
    (controlEventID > currentControlEventID)) or
    controlEventID < currentControlEventID
    ]&(errorMsg:= STATE_ERROR;
      errorMsgID:= controlEventID;
      currentControlEventID:= controlEventID;
      stateReplyID:= controlEventID;  act_PLAYING)
  )
)

act_exit_STREAM_READY = Skip

act_STREAM_READY = (
  (
    [controlEvent = DISCONNECT and
    controlEventID > currentControlEventID
    ]&(currentState:= DISCONNECTED;
      currentControlEventID:= controlEventID;
      stateReply:= DISCONNECTED;
      stateReplyID:= controlEventID;
      errorMsgID:= controlEventID;
      errorMsg:= NO_ERROR;  act_DISCONNECTED)
  )
  []
  (
    [controlEvent = PLAY and
    controlEventID > currentControlEventID]&(currentState:= PLAYING;
      currentControlEventID:= controlEventID;
      stateReply:= PLAYING;
      stateReplyID:= controlEventID;
      errorMsgID:= controlEventID;
      errorMsg:= NO_ERROR;  act_PLAYING)
  )
  []
  (
    [controlEvent = SETUPSTREAM and
    controlEventID > currentControlEventID
    ]&(currentControlEventID:= controlEventID;
      stateReply:= STREAM_READY;
      stateReplyID:= controlEventID;
      errorMsgID:= controlEventID;
      errorMsg:= NO_ERROR;  act_STREAM_READY)
```

```
  )
  []
  (
    [((controlEvent <> SETUPSTREAM) and
    (controlEvent <> DISCONNECT) and
    (controlEvent <> PLAY) and
    (controlEventID > currentControlEventID)) or
    controlEventID < currentControlEventID
    ]&(errorMsg:= STATE_ERROR;
      errorMsgID:= controlEventID;
      currentControlEventID:= controlEventID;
      stateReplyID:= controlEventID;  act_STREAM_READY)
  )
)

act_exit_Initial = Skip

act_Initial = (
  (currentState:= DISCONNECTED;
  currentControlEventID:= 0; act_INIT)
)

@ act_Initial
end
```

3.3 Dwarf example

The example in Figure 3 is the SysML state machine of the Dwarf signal case study. The state machine models the interaction of the dwarf signal through a `setState` signal, which carries the configuration.

The CML model of the Dwarf signal used as a case study for CML only accepts appropriate requests. For instance, it is not possible to request the signal to change from STOP to DRIVE. In a SysML model that follows the semantics in D22.4, it is not possible to refuse an event, and thus it is necessary to accept but ignore any events that were not allowed in the CML model. The synchronous model encoded in S2C-lite overcomes this issue and can refuse requests. As a consequence, it supports better validation of the internal logic of the Dwarf signal example.

The generated model is shown below.

```
channels
  setState : States_t
  switchOff : Lamps_t
  switchOn : Lamps_t

types
  States_t = <dark> | <stop> | <warning> | <drive>
  Lamps_t = <L1> | <L2> | <L3> | <noLamp>

values
  dark = <dark>
  stop = <stop>
  warning = <warning>
  drive = <drive>
  L1 = <L1>
```

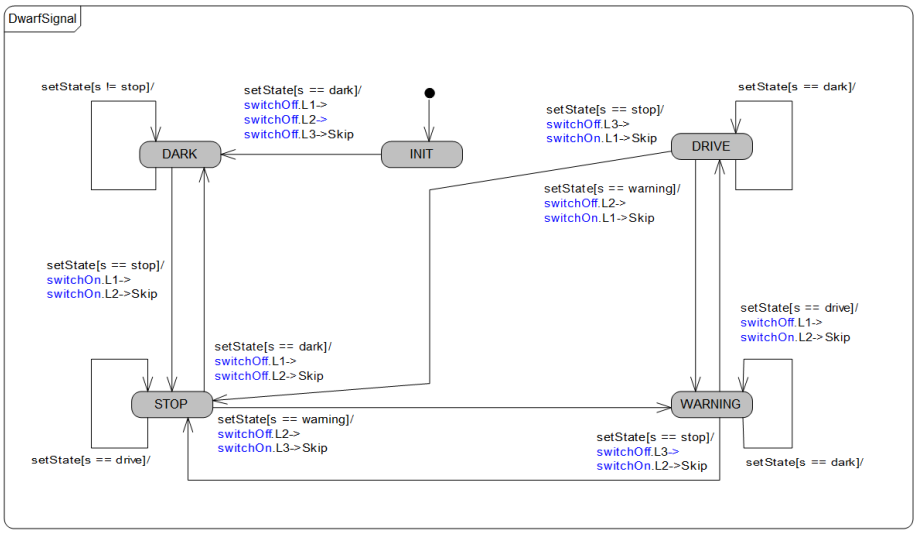

Figure 3: Dwarf signal example

```
  L2 = <L2>
  L3 = <L3>
  noLamp = <noLamp>

types
  float ::
  timer ::
  unsigned_char ::
  unsigned_int ::

class SYSTEM = begin
end

class SystemUnderTest = begin
state
  DwarfSignl : DwarfSignal
end

class DwarfSignal = begin
end

class TestEnvironment = begin
end

process DwarfSignal = begin
actions
  act_exit_DARK = Skip
  act_DARK = (
    (setState?s:(s <> stop) -> act_DARK)
    []
    (setState?s:(s = stop) -> switchOn.L1-> switchOn.L2->Skip ;
        act_STOP)
  )

  act_exit_DRIVE = Skip

  act_DRIVE = (
```

```
    (setState?s:(s = dark) -> act_DRIVE)
    []
    (setState?s:(s = stop) -> switchOff.L3-> switchOn.L1->Skip ;
        act_STOP)
    []
    (setState?s:(s = warning) -> switchOff.L2-> switchOn.L1->Skip ;
        act_WARNING)
  )

  act_exit_INIT = Skip

  act_INIT = (
    (setState?s:(s = dark) -> switchOff.L1->
      switchOff.L2-> switchOff.L3->Skip ; act_DARK)
  )

  act_exit_STOP = Skip

  act_STOP = (
    (setState?s:(s = dark) -> switchOff.L1-> switchOff.L2->Skip ;
        act_DARK)
    []
    (setState?s:(s = drive) -> act_STOP)
    []
    (setState?s:(s = warning) -> switchOff.L2 -> switchOn.L3->Skip ;
        act_WARNING)
  )

  act_exit_WARNING = Skip

  act_WARNING = (
    (setState?s:(s = drive) -> switchOff.L1-> switchOn.L2->Skip ;
        act_DRIVE)
    []
    (setState?s:(s = stop) -> switchOff.L3-> switchOn.L2->Skip ;
        act_STOP)
    []
    (setState?s:(s = dark) -> act_WARNING)
  )

  act_exit_Initial = Skip

  act_Initial = (
    (act_INIT)
  )

@ act_Initial
end
```

3.4 Hybrid SUV

The example in Figure 4 is the state machine of the standard OMG example Hybrid-SUV. It is the largest model we have tested the tool so far, and includes a lot of irrelevant information that have no bearing on our translation (e.g., requirements, constraints). It is also the most complex example, containing composite states.

The generated model is shown below.

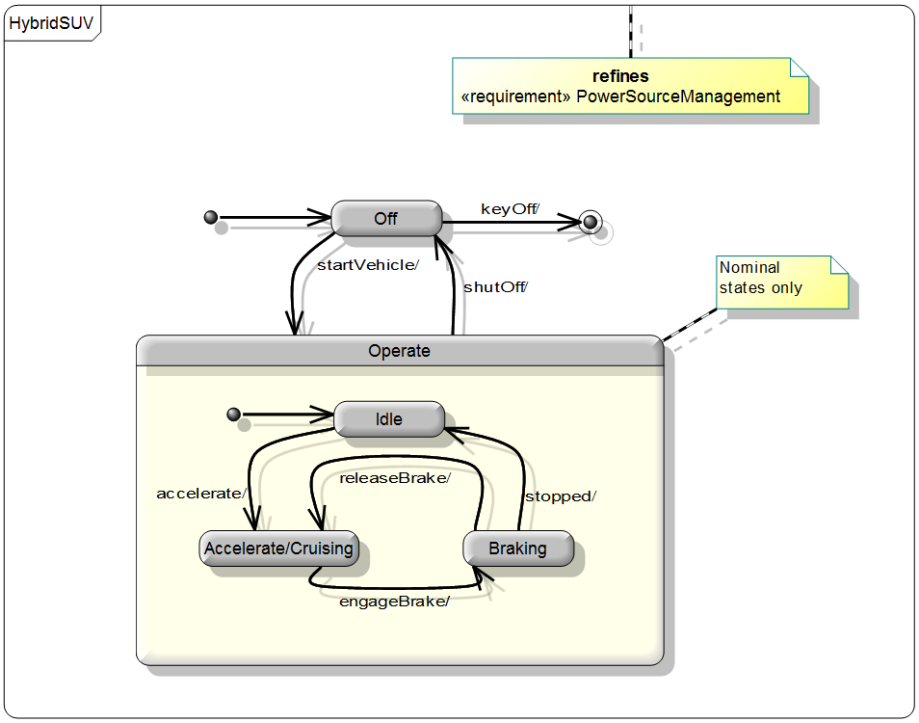

Figure 4: Hybrid SUV example

```
channels
  keyOff
  startVehicle
  shutOff
  engageBrake
  releaseBrake
  stopped
  accelerate

types
  Accel ::
  CostEffec ::
  Dist ::
  Economy ::
  HorsePwr ::
  Press ::
  Price ::
  Temp ::
  Time ::
  Vel ::
  Vol ::
  Weight ::
  ID ::
  PayloadCapacity ::
  ElectricCurrent ::
  Fuel ::
    temperature: Temp
    pressure: Press
  ICEData ::
```

```
    rpm: int
    knockSensor: bool
  Torque ::

class CapacityContext = begin
state
  ad : AutomotiveDomain
end

class EconomyContext = begin
state
  ad : AutomotiveDomain
  delta_t : GlobalTime
end

class GlobalTime = begin
end

class MeasuresOfEffectiveness = begin
state
  HSUValt1 : HybridSUV
end

class UnitCostContext = begin
state
  ad : AutomotiveDomain
end

class accelerator = begin
end

class AutomotiveDomain = begin
state
  HSUV : HybridSUV
  vehicleCargo : Baggage
  drivingConditions : Environment
end

class Baggage = begin
end

class BatteryPack = begin
end

class BodySubsystem = begin
state
  sn : ID
end

class BrakePedal = begin
end

class BrakeSubsystem = begin
state
  sn : ID
  bkp : BrakePedal
end

class CAN_Bus = begin
end

class ChassisSubsystem = begin
```

```
state
  sn : ID
  wha : WheelHubAssembly
end

class Differential = begin
end

class ElectricalPowerController = begin
end

class ElectricMotorGenerator = begin
state
  sn : ID
end

class Environment = begin
state
  weather : Weather
  road : Road
  object : ExternalObject
end

class ExternalObject = begin
end

class FrontWheel = begin
end

class Fuel = begin
end

class FuelInjector = begin
end

class FuelPump = begin
end

class FuelRail = begin
end

class FuelRegulator = begin
end

class FuelTankAssembly = begin
state
  fp : FuelPump
end

class HybridSUV = begin
state
  VIN : ID
  PayloadCapacity : Weight
  FuelEconomy : Economy
  QuarterMileTime : Time
  Zero60Time : Time
  UnitCost : Price
  CostEffectiveness : CostEffec
  bk : BrakeSubsystem
  p : PowerSubsystem
  b : BodySubsystem
  i : InteriorSubsystem
```

```
  l : LightingSubsystem
  c : ChassisSubsystem
end

class InteriorSubsystem = begin
state
  sn : ID
end

class InternalCombustionEngine = begin
state
  sn : ID
  fi : FuelInjector
  fra : FuelRail
  fi1 : FuelInjector
  fi2 : FuelInjector
  fi3 : FuelInjector
  fi4 : FuelInjector
  fre : FuelRegulator
end

class LightingSubsystem = begin
state
  sn : ID
end

class PowerControlUnit = begin
end

class PowerSubsystem = begin
state
  sn : ID
  bkp : BrakePedal
  wha : WheelHubAssembly
  acl : accelerator
  bp : BatteryPack
  ft : FuelTankAssembly
  ecu : PowerControlUnit
  epc : ElectricalPowerController
  dif : Differential
  trsm : Transmission
  ice : InternalCombustionEngine
  emg : ElectricMotorGenerator
  rfw : FrontWheel
  lfw : FrontWheel
  can : CAN_Bus
end

class Road = begin
end

class Transmission = begin
state
  sn : ID
end

class Weather = begin
end

class WheelHubAssembly = begin
end
```

```
class SUV_EPA_Fuel_Economy_Test = begin
state
  TestVehicle1 : HybridSUV
end

process HybridSUV = begin
state
  VIN : ID
  PayloadCapacity : Weight
  FuelEconomy : Economy
  QuarterMileTime : Time
  Zero60Time : Time
  UnitCost : Price
  CostEffectiveness : CostEffec
  active_act_Operate: <act_Accelerate_Cruising> | <act_Braking> |
                      <act_Idle> | <act_Initial1> := <act_Initial1>

actions
act_exit_Final = Skip
act_Final = Stop

act_exit_Off = Skip
act_Off = (
  (keyOff -> act_Final)
  []
  (startVehicle -> act_Operate)
)

act_exit_Operate = (
  cases active_act_Operate:
  <act_Accelerate_Cruising> -> act_exit_Accelerate_Cruising,
  <act_Braking> -> act_exit_Braking,
  <act_Idle> -> act_exit_Idle,
  <act_Initial1> -> act_exit_Initial1,
  others -> Skip
end
)
act_Operate = (act_Initial1)/_\(
  (shutOff -> act_Off)
)
act_exit_Accelerate_Cruising = Skip
act_Accelerate_Cruising = active_act_Operate := <
    act_Accelerate_Cruising>;(
  (engageBrake -> act_Braking)
)
act_exit_Braking = Skip
act_Braking = active_act_Operate := <act_Braking>;(
  (releaseBrake -> act_Accelerate_Cruising)
  []
  (stopped -> act_Idle)
)
act_exit_Idle = Skip
act_Idle = active_act_Operate := <act_Idle>;(
  (accelerate -> act_Accelerate_Cruising)
)
act_exit_Initial1 = Skip
act_Initial1 = active_act_Operate := <act_Initial1>;(
  (act_Idle)
)

act_exit_Initial = Skip
act_Initial = (
```

```
  (act_Off)
)


@ act_Initial
end
```

4 Conclusions

The semantics implemented in S2C-lite imposes a synchronous interpretation of events and supports better analysis via simulation. Furthermore, due to this interpretation and a number of simplification assumptions, the generated code is more readable, and it is easier to trace parts of the CML model to the corresponding elements of the SysML model.

While the generated models can be rather large, this is due to the supporting definitions (e.g., blocks and types), not the state machine. For instance, in the hybrid SUV example in Section 3.4, 180 lines are related to supporting definitions, while the state machine process is 60 lines long. However, when the official semantics of SysML from OMG is used, following the rules defined in Deliverable D22.4 [MCI+13], the models would be an order of magnitude larger. Thus, we believe that the semantics used in S2C-lite is worth considering for any follow-on work to the COMPASS project in the future.

The addition of composite states was necessary to cover the hybrid SUV example, however, it adds complexity to the specification. In particular, each state generates two actions, one that encodes the state and another that encodes the execution of exit actions. This is necessary because the hierarchy of states imposes particular orders of execution of exit actions depending on which substates are active. Additionally, state components must be created to keep a record of which substates are active.

It is worth mentioning that this semantics does not replace that of D22.4, as they serve different purposes. For an initial analysis of individual components, this semantics is adequate as it is more treatable, but for a deeper analysis of the system as a whole, the semantics in D22.4 should be used.

References

[MCI+13] Alvaro Miyazawa, Ana Cavalcanti, Juliano Iyoda, Márcio Cornélio, Lucas Albertins, and Richard Payne. Final Report on Combining SysML and CML. Technical report, COMPASS Deliverable, D22.4, March 2013. Available at http://www.compass-research.eu/.