Specification of Embedded Systems
Summer Semester 2020

Session 7

Automated Model-based Code Generation

Jan Peleska
peleska@uni-bremen.de
Issue 3.0
2020-07-20

Chapter 1

Preface

In this document, the material for **Session 7** of the course **Specification of Embedded Systems** is provided. This session introduces the topic of **automated code generation from SysML models**. You will learn how to write your own code generator for producing C-code which runs in a given domain framework.

This document is structured as follows. Overview

- In Section 2, the utilisation of domain frameworks for code generation is described. We introduce the special framework to be used in this lecture. The general structure of the code generator is explained.
- In Section 3, we describe how to create C^{++} class types from composite (block) types in the SysML model.
- In Section 4, the generation of code for executing SysML state machines is explained.
- As usual, these lecture notes end with questions and exercises in Section 5.

A sample code generator program using the Libxml2 for evaluating XMI-files is provided for this session. Moreover, the files of the domain framework are provided as well. The domain framework has to be compiled and linked together withe the generated code. Please download this material for programming the solutions for the exercises.

Contents

List of Figures

Listings

Chapter 2

Domain Frameworks for Code Generation

2.1 Domain Frameworks

Code generation from models is most efficient, when performed for a well-defined application domain and associated well-defined target platform with HW components, operating system, and software libraries. It is obvious that model-based code generation will not comprise *every* software component running on the target platform:

- If a software component (say, a library like the LIBC or the JDK, an operating system, or a device driver) already exists, why should you create a model for this component and re-generate the code from the model?
- Some types of software are not well-suited for being modelled in SysML: for example, graphical user interfaces are typically created according to the **virtual prototyping** paradigm, using GUI wizards.

As a consequence, model-based code generation usually produces the core applications, while service software (GUI, libraries, database software, communication software, ...) and operating system are integrated with the generated code in a second step. The service and operating system components typically represent a reusable collection of code applied for a whole family of products. Therefore, this collection is called the **domain framework**.

Since we only have PCs or laptops available to run the turn indication controller, our domain framework consists of

Our domain framework

1. Windows, Linux, or MacOS operating system
2. UDP/IP communication library
3. A small library to exchange data between global variables (representing connectors between ports) and the UDP/IP communication library.

2.2 Designing the Domain Framework

It is efficient to design a code generator with a specific domain framework in mind [2]. The design process for the framework proceeds along the following steps.

1. **Identify network connections.** In a distributed system, the controllers involved may communicate over different networks and associated protocols, like

   - Ethernet with TCP/IP
   - Ethernet with UDP/IP
   - CAN
   - FlexRay (used in the automotive domain, see `https://en.wikipedia.org/wiki/FlexRay`)
   - AFDX (Avionic Full Duplex Switched Ethernet; this is used today for inter-controller communication in civil aircrafts, see `https://en.wikipedia.org/wiki/Avionics_Full-Duplex_Switched_Ethernet`).

   The connectors in the model connecting ports to be deployed in HW need to be associated with the network/communication drivers to be used.

   In our turn indication controller example, we will only use UDP/IP.

2. **Identify HW interfaces to peripherals.** Peripheral elements in the operational environment of the target system will not always have bus interfaces: they are often controlled or monitored using discrete I/O

or analogue I/O. There is a general trend, however, to turn peripheral devices into more intelligent "systems on a chip" which have similar interfaces as the controllers themselves.

For example, the smoke detectors in an Airbus A350 aircraft communicate with the smoke detection controllers via CAN, and the fasten seatbelt signs, reading lights and many other peripheral devices in the passenger cabin are controlled by means of an Ethernet-based network.

For the turn indication controller, we will also use UDP/IP to realise the interfaces from/to the peripherals lamps, dashboard, turn indication lever, ignition key, battery voltage indication, emergency switch.

3. **Identify operating system.** Controllers for highly safety-critical tasks typically run on **bare metal**: this means, that powering the system just activates a main program which performs all duties of interface management and task scheduling.[1] For control systems of medium and low criticality, multi tasking operating systems are used, since they allow for the utilisation of simpler application programming paradigms.

   In our example, you will use your Windows or Linux or MacOS as operating system, whatever runs on your PC or laptop.

4. **Identify SW processes.** Processes run in their own address space and have their own resources. This has the advantage, that errors produced in one process cannot affect other processes.

   For our example, all software designated to run on one controller will be executed in the same main process. As a consequence, we do not rely on any scheduling capabilities of the underlying operating system. We just expect Windows, Linux, MacOS to start the main program on a CPU core and leave it there. Apart from that, we only need the UDP/IP communication stack provided by the operating system.

5. **Identify threads.** As an alternative to processes, classifier behaviours can run as threads inside a common main process providing the context. Threads can interact with each other more easily (and much faster) via shared variables, but they can also affect each other in unpredictable ways when addresses are miscalculated.

[1] You can learn how to do this in my lecture on Real-Time Operating Systems Development.

For the turn indication controller, we will use simple C^{++} methods as threads running inside each controller's only main process. Recall that all classifier behaviours are represented as state machines in this course, so threads are C^{++} methods executing SysML state machines in a transition-by-transition manner.

6. **Identify inter-process communication mechanisms.** For embedded systems in general, we distinguish between **inter-process communication** and **intra-process communication**. The former provides mechanisms to exchange data between separate processes, such as shared memory, message queues, shared files, and sockets. The latter provides mechanisms to exchange data between threads running in the same process context, such as shared variables, and local (and, therefore, faster) message queues.

   For the turn indication controller, we will use the UDP/IP protocol with sockets for inter-process communication and shared variables for intra-process communication.

7. **Identify scheduler.** Depending on the underlying operating system, several scheduling strategies are available.[2] For embedded control systems, the main characteristics of schedulers should be

   - low latency – the task to be scheduled next should get the CPU with the shortest possible delay, and
   - high-precision periodic task activation – tasks that should run every `k` ms should get the CPU periodically at the designated point in time with minimal jitter and without any drift.

   For the turn indication controller, each process should get the CPU and keep it permanently until the system is shut down. By using multi-core PCs and a cooperative scheduling strategy with high priority, this can be achieved very nicely in Linux, in Windows and MacOS it is more difficult to ensure that a runnable process is never de-scheduled.

   Threads will be scheduled as method calls from the main process to the C^{++} objects executing classifier behaviour.

[2] Linux, for example, provides the **Completely Fair Scheduler** for normal application scheduling, and a round-robin scheduler, as well as a cooperative scheduler with static priorities for soft real-time applications.

2.3 Designing the Code Generator

With the framework-related design decisions at hand, the code generator design can proceed along the following steps.

1. **Identify HW components.** Which parts of the model should be realised in HW? How are the blocks from which these parts are instantiated identified? Recall that the model also contains parts belonging to the context, i.e. to the operational environment of the target system to be developed. It is therefore useful to identify a sub-model by its root, so that "everything underneath the root" should be deployed somewhere in the target system, and therefore has to be processed by the code generator, as long as the part has to be realised in software. A suitable root could be

   (a) a package inside the model,

   (b) a block inside the model representing the complete target system,

   (c) a model element decorated with a new stereotype, such as ≪TargetSystem ≫[3].

   In our turn indication controller example, the parts to be realised in HW are identified by being

   - parts of block TurnIndicationController[4], and
   - typed by blocks possessing full ports (i.e. "real" HW interfaces).

   This means that we prefer variant (b) for identifying the target system, and, consequently, our code generated main interface is specified as

```
1 extern void generateCode(xmlChar* rootBlockName);
```

   We expect that the target system's root block has a unique name; if this assumption would not be fulfilled, we would use the **qualified name** of the root block, like

```
1 TurnIndicationJP::turnindicationsystem::TurnIndicationController
```

[3]This, however, would mean that the code generator relies on a profile extension of the SysML, where this stereotype has been introduced.

[4]These parts are F:FrontController, DL:DoorController, DR:DoorController, R:RearController.

The qualified name consists of model name, package path, and block name; each element separated by C^{++}-style double colons.

The `generateCode()` function calls auxiliary function

```
static void findMainParts(searchResult_t mainPartsList,
                          xmlNodePtr rootBlock);
```

for looking up the parts underneath the root block that correspond to HW components. It gets the handle of an empty list as input and the XMI node representing the root from where to look up the main parts. It looks for parts that are children of the `rootBlock`, are typed as classes, and which possess full ports. These parts are entered into the list to be processed further by the caller. The caller (i.e. `generateCode()`) then invokes function

```
static void createMainProg(xmlNodePtr part);
```

for each part of this list, and the latter function will create the associated main programs.

When trying out the generated code, we could distribute the software on several PCs, each representing one controller. Alternatively, we can run everything on one PC without changing the software structure: since we are choosing UDP/IP for inter-controller communication, the distribution of software on hardware can be changed just by modifying IP addresses and UDP ports.

2. **Translate primitive types.** There are only a few primitive types pre-specified in UML/SysML (`Integer`, `Boolean`, ...). Therefore, we insert C-style type declarations into a designated header file called `projectTypes.hpp`, using an auxiliary function

```
static void createPrimitiveTypes(void);
```

The type declaration introducing `Integer`, for example, looks like

```
typedef int Integer;
```

3. **Translate enumeration types.** For enumerations, the code generator uses auxiliary function

```
static void createEnums(void);
```

As you can see when studying the code generator, file `codegen.c`, this function creates a list of all XMI nodes typed as `uml:Enumeration`; these nodes are used in the XMI file for starting new enumeration type declarations. For each of these nodes, the function traverses the sub-nodes to find nodes named `ownedLiteral`. These specify the names of an enum literal in node attribute `name`.

The generated enumeration type declarations are also inserted into file `projectTypes.hpp` and look, for example, like

```
typedef enum {
        NEUTRAL,
        LEFT,
        RIGHT
} LeverPosition_t;
```

Observe that function `createEnums()` does not evaluate the `specification` nodes which may reside underneath `ownedLiteral` nodes: there, it is possible to specify specific integer values to be associated with each literal. We did not use this feature in our models, so the first literal defined is always associated with integer value 0, the second with 1, and so on, as in the default case for enum declarations in the C programming language.

No specific int-values for enum literals

Further observe that *all* our type names (also value types and blocks (i.e. class types)) carry unique names, so that we do not have to consider different name spaces during code generation. Of course, for an industrial-strength code generator, the consideration of name spaces is mandatory, but it is quite cumbersome to implement and requires to add many lines of code[5] to the generator that do not really help for increasing the general understanding of code generation.

No specific name spaces

4. **Translate value types.** Value types specified in the model are also inserted into file `projectTypes.hpp` by the code generator, which uses function

```
static void createValueTypes(void)
```

for this purpose. The function first collects all value types (these are variants of blocks in the SysML profile) in a list. For each value type, the function looks up its associated UML `base_DataType` which also

[5] You can believe me, we are doing this in our company, and it's quite a big chunk of extra work.

contains the name of the value type/UML data type. In the programming language, we wish to associate the value type name with its defining type. This is stored in the `generalization` node underneath each `uml:DataType` node. The value types we have specified in the turn indication model are all generalised as integers, so the generated C/C^{++} type declarations look like like

```
1 typedef Integer BatteryVoltage;
2 typedef Integer Duration;
3 typedef Integer LampCurrent;
```

5. **Translation of aggregated types.** More complex types are represented in SysML by blocks and their UML counterpart classes. This is the most complex aspect of type generation and is discussed in the separate Section 3.

6. **Translate ports and connectors.** In our SysML model, ports of parts are connected by UML **Connectors**. We do *not* use additional item flows, though they would be practical to indicate the direction of the data flow. Instead, the flow direction is implicitly determined as follows:

   - The state machine performing assignments to the port in an opaque behaviour (this may also occur in an operation called by the state machine) acts as writer (or source or sender) to the connector.
   - The state machine reading from its port in an opaque expression (guard condition or right-hand side of an opaque assignment) acts as reader (or target or receiver).

   Since we adhere to the rule that at most one write is performed to a port in one transition step (run-to-completion), we can represent connectors as global variables, and the state machine writing to the port performs assignments to the variable, and the state machine reading from its port performs variable reads in the generated code.

7. **Translate behaviour.** Since classifier behaviour is represented by SysML state machines and block operations in our model, the code generator needs to transform state machines into C^{++} code. This is the most complex part of the generation and is explained in the separate Section 4.

Chapter 3

Generation of Block/Class Types

This section is new in Issue 2.

In SysML, aggregated types are created using blocks or interface blocks. Note that these blocks may be distributed over all packages of the model; there is not restriction that they should be declared underneath the root block from where code is generated. Therefore, the `generateCode()` function collects *all* blocks and interface blocks declared anywhere in the model, identifies the associated UML base class declarations and calls function

```
1 static void createClassDeclaration(xmlNodePtr theClass)
```

for each of these base classes, with the XMI node representing the class as input.

Skip existing project types

Function `createClassDeclaration` starts with a check whether the class has already been declared in `projectTypes.hpp`:

```
1 if ( nameListContains(projectTypesList, className) ) return;
```

The type declarations contained there are collected in the list `projectTypesList`, and need not be re-declared as class types. This happens when (interface) blocks are typed by a single value property which is a of primitive type or enumeration type. In this case, the type is already contained in `projectTypes.hpp` and does not need to be re-declared.

Lookup parent classes and ports

Next, it is checked whether the class under consideration has a parent class. In our models, we have applied the restriction that ports should be declared already on the level of the parent class, so they have to be looked up at the parent.

```
1  xmlChar* parentClassName;
2  xmlNodePtr parentClassNode;
3  int hasGeneralisation = findGeneralisation(theClass,
4                                             &parentClassName,
5                                             &parentClassNode);
6
7  int numberOfPorts = 0;
8  searchResult_t portList;
9  iterator_t portIte;
10 ...
11 createNodeList(&portList);
12 portIte = getMyPorts(portList,
13                       ((hasGeneralisation) ? parentClassNode : theClass),
14                       &numberOfPorts);
```

To this end, function `findGeneralisation()` looks up whether a parent exists. If this is the case, 1 is returned, and class name and xml node pointer of the parent class are written to the out parameters `parentClassName`, `parentClassNode` (lines 1—5). Then the list of all ports declared for the class or the parent, respectively, written by function `getMyPorts()` to a new node list `portList` (lines 11—14).

Create header file

The next step is to create a C++ header file for the class to be declared. It gets the same file name as the class, with extension `.hpp`. The classes will be represented by header files only, with method code inserted to the header file. Therefore, no implementation files (`.cpp`-files) need to be created.

Insert include directives

Since classes may need other types, the following include directives are written to the header file.

- The `projectTypes.hpp` file with the primitive types, value types, and enumeration types.
- If a parent class exists, the header file of the parent class.
- The header files of aggregated port types, if they are not already included by the parent.
- The header files of class types associated with parts or other aggregated properties whose type is not already contained in `projecktTypes.hpp`.

Class declaration

Next, the proper class declaration starts. We use the C/C++ keyword `struct` instead of `class` to start this declaration, since `struct`-classes have all their attributes, constructors, and operations declared as `public` by default, and this is what we prefer for generated code: The `private` keyword helps during software development to avoid unwanted access to attributes and methods that might be subject to change later on. In our context, there

is no manual software development, the code is generated directly from the model, so there is no reason for hiding anything to anybody.

For every generated class, the following principle is applied regarding ports and their connectors created when instantiating the block as a part. and connectors.

Port and connector representation principle

1. The outer class interface ports (`CANin, BatVolIN, CurrentOUT, PinProgram` in Fig. 3.1) modelled on the boundary of the block/class need to be bound by references to variables of the surrounding context. If the surrounding context is a main program (this is the case for parts that are instances of the `DoorController` in Fig. 3.1), the main program needs to declare global variables storing the port data.

2. Now consider inner ports of parts inside the class under consideration that are not connected to a boundary port, but are linked by inner connectors of the surrounding class. These inner ports are bound by reference to class attributes representing the inner connectors (ports `switchIn` and `switchOut` and the associated connector in Fig. 3.1).

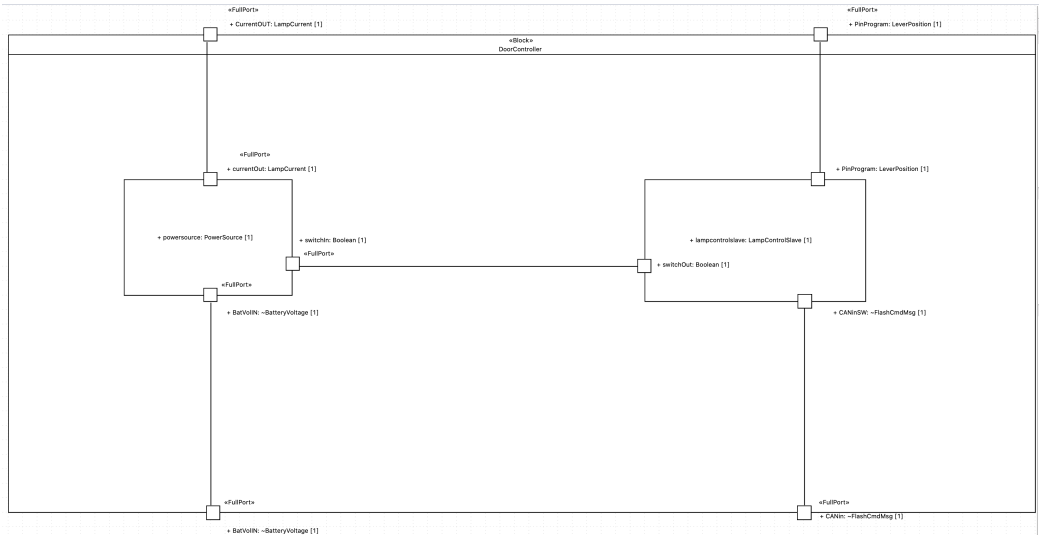

Figure 3.1: Internal block diagram of the door controller.

Therefore, the code generator for a class needs to declare all its inner connectors as attributes. This is done by means of a call to

Declare inner connectors

```
static void declareInnerConnectors(xmlNodePtr theClass)
```

This function traverses the nodes of all children underneath XMI node `theClass`. For each connector found (connectors reside in XML nodes with node name `ownedConnector`), it is checked whether it is an *inner* connector. This is the case when *both* connector ends have attribute `partWithPort` defined. If one connector end does not associate a `partWithPort`, then the port linked to this connector end is a boundary port. The type of the connector attribute is the type of any of its attached ports. The ports at both ends must have the same type. For the example of class `DoorController` shown in Fig. 3.1, just one connector attribute has to be declared:

```
// Declarations of inner connectors
    Boolean Connector9;
```

From the model you can see that this is the inner connector linking ports `switchIn` and `switchOut`.

Declare ports

If the class has its own ports, we declare the interface ports (`CANin, BatVolIN, CurrentOUT, PinProgram` in Fig. 3.1) of the class as C++ references. When the class is instantiated, the concrete variables (connector attributes or global variables) of the context instantiating the class need to be provided as constructor parameters, so that the reference variables can be immediately bound to the variable addresses. For the example from Fig. 3.1, the generated port declarations in header file `DoorController.hpp` looks like this:

```
// Port declarations
    FlashCmdMsg& CANin;
    LampCurrent& CurrentOUT;
    LeverPosition& PinProgram;
    BatteryVoltage& BatVolIN;
```

These declarations are done by function

```
static void declarePorts(searchResult_t portList)
```

which runs through the list of all ports of the class, gets their name and type[1] and writes the declaration in the format shown above.

Declare parts and other properties

Now the part declarations and other property declarations are performed using function

[1] We have auxiliary function `static xmlChar* extractTypeFromNode(xmlNodePtr theNode)` for this purpose.

```
1 static void declareParts(xmlNodePtr theClass)
```

The parts and properties are found in the XMI file as children of the class under consideration. The XMI nodes are named `ownedAttribute`, and their xmi:type is `uml:Property`. Their property type is extracted again using function `extractTypeFromNode()`.

Parts and value properties are represented as "normal" class attributes. For example, the part declarations of class `DoorController` look like this.

```
1 // Part declarations
2     PowerSource powersource;
3     LampControlSlave lampcontrolslave;
```

The most "tricky" part during class generation is the creation of the constructor. This is because

Constructor Declaration

- the constructor needs a parameter list of variables to be bound by reference to the ports of the class,
- the constructor has to call the constructor of a parent class if it exists, and
- the constructor needs to initialise all parts, supplying the variable addresses to be associated with the outer ports of each part.

For class `DoorController`, the generated constructor looks like this.

```
1  DoorController(FlashCmdMsg& CANin,
2      LampCurrent& CurrentOUT,
3      LeverPosition& PinProgram,
4      BatteryVoltage& BatVolIN) :
5      CANin(CANin),
6      CurrentOUT(CurrentOUT),
7      PinProgram(PinProgram),
8      BatVolIN(BatVolIN),
9      powersource(CurrentOUT,Connector9,BatVolIN),
10     lampcontrolslave(Connector9,CANin,PinProgram)
11   { }
```

Each instance of the `DoorController` gets the variable references for storing the port data as parameters in the constructor invocation (`CANin, ..., BatVolIN`). In the first four initialisations following the colon ':', the variable addresses are written to the reference variable used as ports. Do not be confused by both parameter and port names being the same: for example, in initialisation expression `CANin(CANin)`, the first `CANin` denotes the class attribute (i.e. the port reference) to be initialised, and the second `(CANin)` is the first call parameter used in the constructor.

After the four port initialisations, the part initialisations follow, using the part names and providing their constructor parameters:

```
powersource(CurrentOUT,Connector9,BatVolIN),
lampcontrolslave(Connector9,CANin,PinProgram)
```

You can see from Fig. 3.1 that ports of part `powersource` are initialised as follows.

- Port `powersource.currentOut` gets the address of port variable `DoorController.CurrentOUT`.
- Port `powersource.switchIn` gets the address of inner connector attribute `DoorController.Connector9`.
- Port `powersource.BatVolIN` gets the address of port variable `DoorController.batVolIN`.

To see that this matches properly, you can check the constructor of the part's class, `PowerSource`:

```
PowerSource(LampCurrent& currentOut,
    Boolean& switchIn,
    BatteryVoltage& BatVolIN) :
    currentOut(currentOut),
    switchIn(switchIn),
    BatVolIN(BatVolIN)
  { }
```

The constructor declaration is performed by function

```
static void declareConstructor(searchResult_t portList,
                               int hasGeneralisation,
                               int numberOfPorts,
                               xmlNodePtr theClass,
                               xmlChar* className,
                               xmlChar* parentClassName)
```

Furthermore, consider how the main programs initialise the instances with full ports. We show this using the example of the left door controller instance `DL : DoorController` which is created by main program `DL/DL.cpp`.

Part instantiation in the main program

```
1  #include "projectTypes.hpp"
2  #include "DoorController.hpp"
3
4  FlashCmdMsg CANin;
5  LampCurrent CurrentOUT;
6  LeverPosition PinProgram;
7  BatteryVoltage BatVolIN;
8
9  int main(int argc, const char * argv[]) {
10
11   DoorController DL(CANin,CurrentOUT,PinProgram,BatVolIN);
12   ...
13 }
```

You can see that the main program provides the global variables `CANin,...,BatVolIN` to the constructor of the `DoorController` instance `DL`, where they are linked to the ports of the instance by means of the constructor, as explained above.

Chapter 4

Code and Data Generation for SysML State Machines

– This section is new in Issue 3 —

Among the various possibilities to specified classifier behaviour in UML, state machines are the most important ones [3, 14.2.1], as discussed in this course.

4.1 Scheduling Strategy

Before creating a code generator, it has to be decided how conceptually concurrent behaviour should be realised in the target environment. UML/SysML allows for modelling concurrent behaviour by

- specifying activities with concurrent threads represented in so-called **swim lanes** (not covered by this course),
- specifying state machines with concurrent regions, and
- interpreting classifier behaviour executed by different blocks to be concurrent.

For the context of this course, we assume that our target platforms are simple single-core one-board controllers and systems on a chip, where applications can only be scheduled in round-robin fashion according to the concept of **cooperative multi tasking**: applications need to return from operation

execution, in order to give way to the next application to run for a certain number of processing steps.

This serialised behaviour is enforced by our code generator through the creation of main programs executing all state machines deployed is one-by-one fashion in a non-terminating **main loop**. To be more detailed, the main loop (see Listing 4.1 for an example) commands the deployed part to perform a processing step, and the deployed part then passes this command on to all of its sub-parts with own classifier behaviour, see Listing 4.2 for an example.

Each state machine in turn just checks whether it can perform **one run to completion**: from the actual simple state the SM resides in[1], it is checked whether one of the outgoing transitions may be triggered. If this is the case, the resulting **compound transition** is executed, potentially transiting across several pseudo states and leaving/entering different regions until a simple state is reached. Our code generator encapsulates this behaviour in so-called **step functions** that are defined for each state and each region and execute at most one compound transition. After that the operation returns, so that the next state machine can execute within the main-loop cycle.

Listing 4.1: Main-loop of the generated main-function for part `R` of type `RearController`.

```
1 while(1) {
2     commandReceptionR;
3     R.step();
4     dataTransmissionR;
5 }
```

[1]For concurrent state machines, the SM can reside in several simple states simultaneously, then each of these states are checked for possible transitions.

Listing 4.2: Invocation of step functions for parts of class `RearController`.

```
1  struct RearController
2  {
3  ...
4  // Part declarations
5      CANdriverSM candriver;
6      ControlLogicSM controllogic;
7      LampControlSlaveSM lampCtrlSlaveR;
8      LampControlSlaveSM lampCtrlSlaveL;
9      PowerSourceCtrlSM pwrRightLamp;
10     PowerSourceCtrlSM pwrLeftLamp;
11     PinProgramBlockLeft pinprogramblockleft;
12     PinProgramBlockRight pinprogramblockright;
13 ...
14 void step() {
15     candriver.step();
16     controllogic.step();
17     lampCtrlSlaveR.step();
18     lampCtrlSlaveL.step();
19     pwrRightLamp.step();
20     pwrLeftLamp.step();
21     pinprogramblockleft.step();
22     pinprogramblockright.step();
23 }
24 ...
25 };
```

4.2 Finding Blocks With Classifier Behaviour

Before generating code for state machines, it is necessary to know from where to invoke this code. To this end, we need to find blocks whose UML base classes

1. are declared as *active* by setting class attribute `isActive` to `true`,
2. are associated with ta classifier behaviour by setting class attribute `classifierBehavior` to the xmi:id of the state machine modelling this behaviour, and
3. have an owned behaviour with `xmi:type="StateMachine"`, whose `xmi:id` equals the one specified in the classifier behaviour.

We exploit these facts is a "reversed" fashion:

- First we look for *any* state machine in the model, and then

- we look for the class possessing this machine as classifier behaviour.
- The associated SysML-block is disregarded: all its behavioural details are represented by its base class anyway.

Restrictions

UML/SysML allows for state machines being used to specify, for example, the behaviour associated with an operation or with an activity node. Such state machines are disregarded by our generator. Our state machines, however, may posses hierarchic nodes or parallel regions containing subordinate state machines.

4.3 State Machine Generator – Software Structure

The overall structure of the state machine code generator is as follows.

Invocation of the SM Generator. State machine generation is invoked from the main function `generateCode()` of the code generator, as shown in Listing 4.3. A loop over *all* state machines of the model is performed (Ziele 5—7). For each machine, their SM generation function `createSmDeclaration(theSm)` is invoked with the XMI-node representing the SM as input.

Listing 4.3: Invocations of the state machine generator.

```
1 // C++ Class types derived from state machines
2 searchResult_t smList;
3 iterator_t smIte = findNodesByType(&smList, (xmlChar*)"uml:StateMachine");
4 xmlNodePtr theSm;
5 while ( (theSm = nextSearchResult(smList, smIte)) ) {
6     createSmDeclaration(theSm);
7 }
```

In the next paragraphs, the main steps performed by `createSmDeclaration(theSm)` are explained.

Check for classifier behaviour. The state machine's parent is checked, whether it is a class which has the SM as classifier behaviour. If the SM is

just a sub-machine to another state, it is simply skipped, since its code will be generated together with the higher-level SM. All other state machine will be reported by an error message, and they are also skipped by the SM code generator.

SM class declaration. All remaining state machines have a parent class, so they are declared as sub-classes of the parent.

Attributes. Three kinds of auxiliary attributes are declared by recursively visiting (depth-first search) all XMI nodes below `theSm` and applying a function for producing declaration code. This is shown in Listing 4.4.

Listing 4.4: Declaration of auxiliary attributes by the SM code generator.

```
1 recursiveSubNodeActions(theSm,createStateNumDeclaration);
2 recursiveSubNodeActions(theSm,createEnterStateTimerDeclaration);
3 createNameList(&oldEventNameList);
4 recursiveSubNodeActions(theSm,createChgCndStatusVar);
```

- The first kind of auxiliary attributes represents internal numbers for simple states, one attribute per SM region.
- The second kind stores time stamps of the most recent entry of a (simple or hierarchic) state. It is associated with a flag indicating whether an associated time event has already occurred.

  With respect to time events, we only allow one transition per state to be triggered by a time event. This fits for many applications. However, there are sophisticated cases where transitions with time events are additionally guarded by conditions about data. Then it can be the case that the transition whose time event occurs first cannot be taken, because its guard condition evaluates to `false`. Then another timed transition, whose time event elapses later could be taken. This situation is not yet covered by the code generator. Restriction
- The third kind of attributes consists of Booleans stating whether the last evaluation of a change condition resulted in `true` or `false`. One attribute per change condition is introduced.

In the next paragraphs, it will become clear how these attributes are used. As an example, Listing 4.5 shows the attributes generated for state machine `ControlLogicSM` in our turn indication model.

Listing 4.5: Resulting attribute declarations in generated file `ControlLogicSM.hpp`.

```
1  ...
2  struct ControlLogicSM : public ControlLogic {
3  // State number attributes for each region used by the SM
4      size_t stateNumRegion1;
5      size_t stateNumRegion1LRFlashingLRFlashingRegion1;
6
7  // Last entry-to-state time stamps and "timer elapsed" flags
8      unsigned long long entryTimerEMERGENCY_FLASHING;
9      bool entryTimerEMERGENCY_FLASHINGElapsed;
10     unsigned long long entryTimerNO_FLASHING;
11     bool entryTimerNO_FLASHINGElapsed;
12 ...
13 // Last evaluation results for change conditions associated
14 // with change events
15     bool IgnOffEventOld;
16     bool EmerOnEventOld;
17     bool toLEventOld;
18 ...
```

Constructor declaration. Since our SMs implement owned behaviour of some block/class, they are realised as C++ sub-classes, so that they automatically find all symbols (attributes, operations) of the parent class in their scope. The constructor of the parent class needs to be called, and the references to ports need to be passed on to the paren. In the constructor body, an initialisation function is called which activates the enter-region operations for the top-level region(s), as described below.

Operations of the SM class – an overview. When implementing an SM in C++ or some other programming languages, the following variants of class operations are needed, since these are directly induced by the behavioural semantics of UML/SysML state machines, as explained in this course and described in [1, 3].

Enter-region operations. The immediate sub-structure of a state machine consists of one region for sequential state machines or several regions in case of parallel state machines with so-called **orthogonal** states.

Moreover, states associated with sub-machines realising do-actions are associated with lower-level regions. When the SM starts to run, its initial region(s) to be entered have to be identified, and these regions need to be entered by executing the chain of pseudo states from the initial pseudo state to the first non-pseudo state. If the latter is a hierarchic state with associated sub-machine behaviour, then the underlying behaviour needs to be activated as well. The

```
1 enter_<region-name>()
```

methods are created by the code generator to initiate these initial steps (more details about this below), and they are re-used during SM operation when one region is left and another is entered.

Enter-state operations. When a state which is not a pseudo state is entered, its entry action has to be executed, and a do-action implemented in a sub-machine might have to be entered, the latter also involving the entry into another region. The

```
1 enter_<state-name>()
```

methods are created by the code generator to manage the state entry and the potential entry into a sub-machine with its regions.

Step operations for SMs. The main program described above activates the "global" step operation of the part deployed by the main program. The latter delegates this command to the step operations of the state machines, as shown in Listing 4.2 above. These operations delegate step processing to the SM's top-level regions who in turn delegate the processing step to sub-regions or to the current simple state the SM resides in.

Step operations for regions. The step operations for regions operate according to the following pattern:

- If a simple state *inside* the region is active (to this end, the current state numbers are evaluated), the state's step function is activated.
- If a state *outside* the region is active, the processing steps of the existing sub-regions are activated.

In Listing 4.6, an example of a region step function is shown.

Step operations for states. The step operation of a simple state checks all transitions by calling their trigger-transition operations. As soon as a transition could fire, the state's step function terminates. The trigger-transition operation determines the new state reached by the compound transition. If no transition emanating from the simple state could fire, it is checked whether a transition emanating from a higher-level state could fire. This check is performed by calling a trigger-transition operation for higher-level states.

There are no step functions for hierarchic states, since steps from these are always incorporated in the simple states' step function. Moreover, no step functions for pseudo-states exist, since these are only visited during a run-to-completion, and no time passes while residing in pseudo-states.

Listing 4.7 gives an example of a step function for a simple state which includes the potential firing of higher-level transitions.

Trigger-transition operations. The trigger-transition operations follow the naming schema

```
1 trigger<transition-name>()
```

For checking whether the transition can fire,

- the guard condition,
- a trigger defined by a signal event, or
- a trigger defined by a change event, or
- a trigger defined by a time event need to be evaluated.

If the transition can fire, the associated compound transition needs to be executed, finally ending up in another simple state. This is the most complex part of the generator, and it is described in more detail below.

Trigger-transition operations for higher-level transitions. The trigger-transition function for a transition emanating from a higher-level state depends on both the lower-level simple state and the transition's higher-level source state. The former dependency occurs since the simple state's exit action needs to be executed first before leaving the higher-level state. Therefore, the naming schema for these operations is

```
trigger<transition-name>From<simple-state-name>()
```

This means that each sub-ordinate state requires a specific trigger-transition function for each higher-level transition. Observe that there can be several levels of hierarchic states above the simple state. Consequently, several exit actions need to be executed, and it is non-trivial to decide which regions are left and which are entered during the compound transition. Therefore, this is also described in more detail below.

Listing 4.6: Generated step function of the top-level regionin SM `ControlLogicSM`.

```
void step_Region1() {
    // Process state machine state from current state
    switch(stateNumRegion1) {
        case 10: step_EMERGENCY_FLASHING();
            break;
        case 12: step_NO_FLASHING();
            break;
        default:
        // Actual active simple state must be in lower-level region
        // underneath hierarchic state LRFlashing
            step_Region1LRFlashingLRFlashingRegion1();
            break;
    }
}
```

Listing 4.7: Generated step function for simple state `TipFlashingCompleted` in SM `ControlLogicSM.LRFlashing`.

```
bool step_TipFlashingCompleted() {
// Try stransitions emanating from this simple state
    if ( triggerleverPositionNeutralTrans2() ) return true;
    if ( triggerignOffTransFromTipFlashingCompleted() ) return true;
// Try higher-level transitions
    if ( triggeremerOverrideTransFromTipFlashingCompleted() ) return true;
    if ( triggerLtoRchangeTransFromTipFlashingCompleted() ) return true;
    if ( triggerRtoLchangeTransFromTipFlashingCompleted() ) return true;
    return false;
}
```

4.4 Generation of Trigger-Transition Operations

4.4.1 Trigger Functions for Transitions Starting in Simple States

The trigger-transition functions are generated by traversing again all subnodes of the SM-node and calling the code generation function for transitions which is called `createTriggerTransFunction()` (see Listing 4.8) and takes an XMI transition node as input. Transitions emanating from pseudostates and transitions emanting from composite (e.g. hierarchic) states are skipped by this function: only transitions starting from simple states are processed. All pseudo states are "encountered" along the way while processing the compound transition starting in this simple state. For transitions emanating from higher-level states, a separate generator function `createTriggerTransFunctionFromSimpleState()` is called as described below.

Listing 4.8: Generator for trigger-transition functions (transitions start in simple state).

```
1  static void createTriggerTransFunction(xmlNodePtr trans) {
2      ... skip transitions that do not start from simple states ...
3
4      xmlChar* transName = getName(trans);
5      fprintf(currentClassHeaderFile,"\n  bool trigger%s() {\n",transName);
6      fprintf(currentClassHeaderFile,"\tbool guard = true;\n");
7      xmlChar* guardCondition = getGuardCondition(trans);
8      if ( guardCondition ) {
9          fprintf(currentClassHeaderFile,"\tguard = %s;\n",guardCondition);
10     }
11     handleTimeEvents(trans, source);
12     handleChangeEvents(trans,source);
13     handleSignalEvents(trans,source);
14     fprintf(currentClassHeaderFile,"\treturn false;\n  }\n");
15 }
```

The trigger-transition function is declared by the generator (line 5), and then a Boolean variable `guard` is declared for carrying the valuation of the guard condition associated with the transition. If a guard condition is missing in the model, this is interpreted as *"the guard always evaluates to* `true`*"*,

so we initialise `guard` with true (lines 6). If an opaque expression has been specified for the transition in the model, the code assigning this Boolean expression to `guard` is generated (lines 7—10), so that "real" guard conditions are evaluated and assigned to `guard` at runtime when the trigger-trans function is called.

Having handled transition guards, the transition triggers need to be processed. This is done in three sub-functions (line 11—13) for the trigger types

- time event,
- change event, and
- signal event.

Restrictions

For signal events, only atomic signals are permitted – signal parameters are not allowed. Moreover, we do not consider **completion events** which are generated after entering a state and completing its entry action and do activity (if any). Completion events trigger transitions without visible triggers [3, 14.2.3.8.3].

handleSignalEvents()

To explain how the transition execution code is generated, we use function `handleSignalEvents(trans,source)` as an example (see Listing 4.9), the other functions operate in a similar way.

Listing 4.9: Generator for transition execution code with signal triggers.

```
1  static void handleSignalEvents(xmlNodePtr trans, xmlNodePtr simpleSource) {
2      xmlNodePtr n;
3      for ( n = trans->children; n; n = n->next ) {
4          ... skip all nodes that do not specify a signal event ...
5          xmlNodePtr signal = findNodeById(getAttr(event,"signal"));
6          if ( ! signal ) continue;
7          xmlChar* signalName = getName(signal);
8          fprintf(currentClassHeaderFile,
9            "\tif ( guard && (%sReception != %s) ) {\n",
10           signalName,signalName);
11         fprintf(currentClassHeaderFile,
12           "\t\t%sReception = %s;\n",
13           signalName,signalName);
14         xmlNodePtr source = findNodeById(getAttr(trans,"source"));
15         generateTransitionExecutionCode(trans,source,simpleSource,0);
16         fprintf(currentClassHeaderFile,"\t}\n");
17     }
18 }
```

Function `handleSignalEvents` loops over all children of the transition until it finds a signal event used by `trans` as trigger. The code generator implements (atomic) signals as counters: Sending a new signal increments the counter. The signal reception attributes of a receiving block in the model are translated to `int`-counters with name

```
1 <signal-name>Reception
```

These are declared in the parent class representing the block whose classifier behaviour is given by the state machine currently generated. A new signal event has occurred when the event number stored in the reception counter differs from the signal counter. The `if`-condition evaluating both the guard and the trigger is generated in lines 8—10. If the conjunction of guard and trigger condition evaluates to `true`, the compound transition associated with `trans` has to be executed. This is done by function `generateTransitionExecutionCode()` which we explain next.

generate-Transition-ExecutionCode()

The function uses 4 input parameters (see Listing 4.10):

1. the transition whose execution code has to be generated,
2. the source state of the transition which may also be a compound state,
3. the simple state where the compound transition execution started, and
4. a flag indicating whether the region containing the simple state has already been left.

The function starts by extracting the target state from the transition (line 5). Then the **least common ancestor (LCA)** of `source` (the transition source) and `target` is determined (line 7): `lca` is the lowest compound state in the state hierarchy containing both `source` and `target`. The LCA is needed to determine which exit actions have to be performed.

- In the first call to `generateTransitionExecutionCode()` for a new compound transition, code for the execution of all exit actions from `simpleStateSource` and its ancestor states which are still **below** `lca` needs to be generated.
- In consecutive recursive calls to `generateTransitionExecutionCode()` for the same compound transition, `source` is a pseudo state and no longer a parent of `simpleStateSource`. In that case, code for all exit actions above `source` whose states are still below `lca` needs to be generated.

Listing 4.10: Generator for transition execution code.

```
1  static void generateTransitionExecutionCode(xmlNodePtr trans,
2                                               xmlNodePtr source,
3                                               xmlNodePtr simpleStateSource,
4                                               int haveLeftSourceRegion) {
5      xmlNodePtr target = findNodeById(getAttr(trans, "target"));
6      if ( !target ) goto lastCmd;
7      xmlNodePtr lca = leastCommonAncestor(source,target);
8      xmlNodePtr p = (isSubStateOf(simpleStateSource,source) ?
9                      simpleStateSource :
10                     source);
11     for ( /* p initialised above */ ; p && p != lca; p = p->parent ) {
12        if ( ! isNodeType(p,"subvertex") ) continue;
13        xmlChar* exitAction = getExitAction(p);
14        if ( exitAction ) {
15            fprintf(currentClassHeaderFile,"\t\t%s\n",exitAction);
16        }
17     }
18     xmlNodePtr sourceRegion = getRegion(simpleStateSource);
19     xmlNodePtr targetRegion = getRegion(target);
20     if ( sourceRegion != targetRegion && ! haveLeftSourceRegion ) {
21         haveLeftSourceRegion = 1;
22        fprintf(currentClassHeaderFile,"\t\tstateNum%s = UNDEFINED_STATE;\n",
23                fullRegionName(sourceRegion));
24     }
25     xmlChar* effect = getTransitionEffect(trans);
26     if ( effect ) {
27         fprintf(currentClassHeaderFile,"\t\t%s\n",effect);
28     }
29     xmlChar* targetType = getXmiType(target);
30     if ( !targetType ) goto lastCmd;
31
32     if (  isXmiType(target,"uml:Pseudostate") ) {
33         if ( isNodeType(target,"connectionPoint") ) {
34             compositeTransFromConnectionPoint(target,
35                                               simpleStateSource,
36                                               haveLeftSourceRegion);
37         }
38         else {
39             compoundTransFromPseudoState(target,
40                                          simpleStateSource,
41                                          haveLeftSourceRegion);
42         }
43     }
44     else {
45         fprintf(currentClassHeaderFile,"\t\tenter_%s();\n",getName(target));
46     }
47 lastCmd:
48     if ( isXmiType(source,"uml:State") )
49         fprintf(currentClassHeaderFile,"\t\treturn true;\n");
50 }
```

The distinction whether to start from the original `simpleStateSource`

or from a pseudo-state `source` is made in line 8—10. The code for executing these exit actions is generated in lines 11—17 by copying the opaque behaviour code of these actions into the transition execution code.

Next, it has to be decided whether `trans` leaves the region where `simpleSourceState` resides in and enters another (line 18—24). To this end, the regions of `simpleSourceState` and `target` are compared. If the former region is left, its state number denoting the active simple state needs to be set to 'undefined'.

In line 25—28, the code for the transition effect is generated by copying the opaque behaviour specifying the transition effect into the transition execution code.

The code generation for a compound transition needs to be continued if the target state reached is a pseudo state or a connection point. This is handled in line 32—43, where sub-functions are called (line 34 and 39), creating code for the evaluation of guard conditions deciding which follow up transitions can be taken from the pseudo state. Applying the depth-first search principle, these sub-functions call `generateTransitionExecutionCode()` again to create code for the transition execution of each transition leaving the pseudo state or connection point.

If `generateTransitionExecutionCode()` reaches a "real" state of type `uml:State`, the recursion ends by calling the enter-state function for `target` (line 45). After that, the execution code for this compound transition has been completely generated, and it returns `true`, because a compound transition has been taken.

4.4.2 Trigger Functions for Transitions Starting in Composite States

This function `createTriggerHigherLevelTransFunctionFromSimpleState()` generates code for executing transitions leaving a compound state, with a specific simple state underneath. It operates in analogy to `createTriggerTransFunction()`.

As an example, we consider the generated code for a compound transition starting from lower-level state `SendLRFlashCmd`: let's analyse the code for handling the firing of higher-level transition `emerOverrideTrans` while residing in that state, see Listing 4.11. No guard condition is specified for transition `emerOverrideTrans` (please look this up in the model, state ma- Example 1

chine diagram `ControlLogic`). Therefore, the `guard` variable is just set to `true` in line 2 and not changed after that. For evaluating the occurrence of the `EmerOnEvent`, the change condition `EmerSwitchPressedINSW` needs to be evaluated. This is done in line 3. In line 4, the occurrence of the change event is checked: The event occurs if

- the old value of the `EmerOnEvent` (i.e., the old value of `EmerSwitchPressedINSW`) was `false`, and
- the new value is `true`.

If the event occurs the current region is left: this is recorded by setting the state number variable to UNDEFINED in line 6. The transition's target state is again a simple state, so its enter-state function is called, and the rigger-transition function returns with `true`. If the event did not occur, the function returns with `false`.

In any case, the current status of the `EmerOnEvent` is recorded in `EmerOnEventOld` (line 5 and 12).

Listing 4.11: Generated trigger-transition method for higher-level transition.

```
1  bool triggeremerOverrideTransFromSendLRFlashCmd() {
2      bool guard = true;
3      bool EmerOnEvent = (EmerSwitchPressedINSW);
4      if ( guard && (! EmerOnEventOld) && EmerOnEvent ) {
5          EmerOnEventOld = EmerOnEvent;
6          stateNumRegion1LRFlashingLRFlashingRegion1 = UNDEFINED_STATE;
7          if ( true ) {
8              enter_EMERGENCY_FLASHING();
9          }
10         return true;
11     }
12     EmerOnEventOld = EmerOnEvent;
13     return false;
14 }
```

The following example shows the code for more complex compound transition executions, see Listing 4.12. Consider transition `leverPositionNeutralTrans2` emanating from simple state `TipFlashingCompleted`. Example 2

Listing 4.12: Generated trigger-transition method for compound transition.

```
1  bool triggerleverPositionNeutralTrans2() {
2      bool guard = true;
3      bool leverToNeutralEvent = (LeverPositionINSW == NEUTRAL);
4      if ( guard && (! leverToNeutralEventOld) && leverToNeutralEvent ) {
5          leverToNeutralEventOld = leverToNeutralEvent;
6          stateNumRegion1LRFlashingLRFlashingRegion1 = UNDEFINED_STATE;
7          if ( EmerSwitchPressedINSW ) {
8              enter_EMERGENCY_FLASHING();
9          }
10         else if ( ! EmerSwitchPressedINSW ) {
11             if ( ! IgnitionOnINSW ) {
12                 enter_NO_FLASHING();
13             }
14             else if ( IgnitionOnINSW ) {
15                 if ( LeverPositionINSW == NEUTRAL ) {
16                     enter_NO_FLASHING();
17                 }
18                 else if ( LeverPositionINSW != NEUTRAL ) {
19                     enter_LRFlashing();
20                 }
21             }
22         }
23         return true;
24     }
25     leverToNeutralEventOld = leverToNeutralEvent;
26     return false;
27 }
```

Again, the guard condition is just `true`. The trigger is given by another change event with change condition `(LeverPositionINSW == NEUTRAL)`. If this change event occurs, the code in lines 5—23 is executed. The region is left via pseudo state `ExitLRFlashing`. This cannot be seen in the code, since this pseudo state is traversed without any visible effect (see model). Then choice state `ifEmerOn` is reached, and the compound transition branches:

- If `EmerSwitchPressedINSW`, simple state `EMERGENCY_FLASHING` is entered (line 8).
- Otherwise we traverse further choice states, either entering simple state `NO_FLASHING` or entering hierarchic state `LRFlashing`. The latter case cannot happen, since the compound transition was triggered by a change event indicating that `(LeverPositionINSW == NEUTRAL)`. The new state number is set and the entry actions are performed for state `NO_FLASHING` by its enter-state function.

Chapter 5

Questions and Exercises

5.1 Questions

5.1.1 Code Generation for a given Domain Framework

Give at least two reasons why it is more efficient to design a code generator for a specific domain framework.

5.1.2 Programming Paradigms

Why do multi-tasking operating systems allow for simpler application programming paradigms than bare metal programming?

5.1.3 Value Types

Consider the value types introduced in our model. All of them are basically `Integer` types. Please explain the advantages of working with value types: why don't we simply use `Integer` instead?

5.1.4 Shared Variables for Ports and Connectors

In Section 3, it has been explained how shared variables or class attributes are declared to store data passed accross ports and connectors. This implies a restriction for the behavioural model parts. Which restriction is this? How could we alternatively represent ports and connectors, so that this restriction could be ignored?

5.2 Exercises

5.2.1 Function createEnums()

Please study the code of function `createEnums()` in file `codegen.c` and insert comments showing that you have understood what happens in this function.

Note. A typical exam question would be *"Please go step-by-step through this function of the code generator and explain what happens and why."*.

5.2.2 Lookup Function for Main Parts

Please study the code of function

```
1 static void findMainParts(searchResult_t mainPartsList,
2                           xmlNodePtr rootBlock);
```

in file `codegen.c` and insert comments showing that you have understood what happens in this function.

5.2.3 Programming: Function getConnectorType()

Please study the code of functions `createMainProgs()` and `declareConnectors()` and insert comments as in the exercises above. The second function has the task to declare global variables corresponding to connectors. These variables are inserted at the beginning of the main program associated with each part to be realised in HW (fron controller, rear controller, ...). The file writes are performed in `declareConnectors()`. In order to declare the variable, however, the *type* associated with the ports linked by the connector needs to be retrieved. This is done in auxiliary function

```
1 static xmlChar* getConnectorType(xmlNodePtr connectorNode);
```

Please program the body of this function. You may assume that the ports at both ends of a connector have the same type, so you only need to investigate one port type.

5.2.4 Understanding Function declareConstructor()

Please study the code of function `declareConstructor()` and insert comments so that you can explain what this function does for declaring a con-

structor and why all these steps are necessary.

5.2.5 Programming: Find Node Containing Classifier Behaviour

Some blocks own a classifier behaviour which is specified in our models by means of state machines. Please write a C function which inputs an XML node pointer to a SysML block and returns NULL if this block does not have classifier behaviour. If the block is directly associated with a classifier behaviour, an XML node pointer to the corresponding state machine should be returned.

Bibliography

[1] Sanford Friedenthal, Alan Moore, and Rick Steiner. *A Practical Guide to SysML, Third Edition: The Systems Modeling Language*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2014.

[2] Steven Kelly and Juha-Pekka Tolvanen. *Domain-Specific Modeling*. JOHN WILEY & SONS, INC., 2008.

[3] Object Management Group. OMG Unified Modeling Language (OMG UML), version 2.5.1. Technical report, OMG, 2017.