Specification of Embedded Systems
Summer Semester 2020

Session 6
Automated Static Model Analaysis

Jan Peleska
peleska@uni-bremen.de
Issue 1.0
2020-06-22

Chapter 1

Preface

In this document, the material for **Session 6** of the course **Specification of Embedded Systems** is provided. This session introduces the topic of **static model analysis**. You will learn how to write your own program for evaluating aspects of static model semantics.

This document is structured as follows. Overview

- In Section 2, the standardized XMI-format for representing SysML models in textual for is described.
- In Section 3, the Libxml2 library is presented. This library will be used for automatted interpretation of SysML models represented in XMI-format.
- In Section 4, we give several examples how static model checking is performed.
- As usual, these lecture notes end with questions and exercises in Section 5.

A sample program using the Libxml2 for evaluating XMI-files is provided for this session. Please download this for programming the solutions for the exercises.

Contents

List of Figures

Listings

Chapter 2

SysML Model Representation in XMI Format

2.1 The XMI Format

UML and SysML are modelling languages containing both structured textual and graphical information. The graphical information, however, only visualises the textual content, so it does not add anything to the semantic meaning of models.

Model serialisation

By **model serialisation**, we mean the transformation of models into one textual document. For UML/SysML, this means *"transformation of everything but the diagrams into one text document"*. For UML/SysML, the well-known XML-format[1] has been chosen as a syntactic framework for representing models and profile specifications. This specialised application of XML is called **OMG XML Metadata Interchange (XMI)**, see [4, G.3] and the XMI specification [2].

Please note that it is not necessary for you to read these normative documents, since we will not need detailed XML-related knowledge in this course, and everything will be explained from scratch for the more specialised XMI format used for model serialisation. Just recall that one of the main objectives of XML is to represent structured documents in standardised form, without losing any structural information. This is exactly what we need for textual representation of UML/SysML models, since we which to specify that a block instance is part of another block, or that a property belongs to

[1] see `https://www.w3.org/TR/REC-xml/`

specific block, or a state to a state machine etc.

The XMI format has been created with several objectives in mind.

Objectives of XMI

Model interchange. Models created with one tool should be readable by another tool, when exported to XMI.

Model checking. The verification of model properties with additional tools should be facilitated by providing a standardised textual format.

Code Generation. The generation of

- simulation code,
- code deployable in the target system,
- test procedures for model-based testing (MBT)

should be facilitated.

Model transformations. The transformation of one model into another (e.g. a platform independent model (PIM) into a platform-specific model (PSM)) should be facilitated by allowing the transformations to read and write the same standardised format.

This model-interchange objective has never been quite fulfilled until today, since tool vendors often make quite a bad job when creating XMI files, with insufficient compliance to the XMI standard. In this respect, Papyrus is very good, since the models you create with this tool are already stored in XMI format: just open the `*.uml` file of a Papyrus project in the Eclipse workspace with an editor, and you will see the XMI representation of the model.

Papyrus stores models in XMI format

2.2 Some Basic Information About UML Profiles

You may recall that the SysML is a **profile** of the UML. Intuitively speaking, this means that SysML introduces new language elements *by explaining their meaning using UML*. As a consequence, a new UML profile never extends the expressive power of the UML, but specialises it and extends its syntax, to facilitate modelling for certain application domains.

A thorough explanation of the profile mechanism is given in [3, 12.3] and [1, 15]. We will describe here only the few details that are important to understand the XMI file structure for representing SysML models.

Stereotypes

The main mechanism for creating new language elements in a profile and explaining them by means of UML is the creation of **stereotypes**. A stereotype is a type[2] of UML language elements extending the UML language element type `Class`. Since it's a class, stereotypes have names, attributes and operations that are specific to the new type of language elements they create. Moreover, each stereotype has a reference to the UML language element type it specialises.

Example: ≪Requirement≫

For example, a SysML requirement is a stereotype named ≪Requirement≫ which is again a subclass of the SysML stereotype ≪AbstractRequirement≫ (since each stereotype is a `Class`, it can also be a sub-class of another stereotype). The ≪AbstractRequirement≫ has attributes `id`, `text`, `derived`, `satisfiedBy`, ..., that we know from working with requirements in the Papyrus turn indication model. Moreover, the ≪AbstractRequirement≫ specialises the more general UML language element type `NamedElement`, so it has a name, an associated namespace (for referencing the requirement by name), and a visibility (public, private, ...). You can gather this information from the SysML standard [4, 16.3.2].

I hope that you can see from this example that the expressiveness of the UML has not been extended (or otherwise changed) by introducing these stereotypes. We have just introduced a special kind of `NamedElement`, so that all instances of this special `NamedElement` have attributes `id`, `text` etc.

Example ≪Block≫

As another example, consider the most widely used SysML stereotype ≪Block≫. As one can see from the SysML standard, a block refers to `Class` itself as the UML language element to specialise. This means, that a block is just a UML class with a name that engineers like better.[3]

[2] a so-called **meta-type**

[3] Engineers want block diagrams, not class diagrams. The latter term sounds too much like software, and software is for sissies. Real engineers deal with HW, and SW is considered as a minor detail they will take care of during the weekend. By the way: real engineers will become extinct by the end of this decade: already today, the software development and verification/validation for complex cyber-physical systems requires more effort than the hardware development.

2.3 Basic Structure of XMI Files

With the explanations of the previous sections in mind, the following choice for the main structure of SysML model serialisation files using XMI is not surprising.

XML document node. Each XMI file starts with an XML node declaring that the file is an XML file:

XML declaration node

```
1 <?xml version="1.0" encoding="UTF-8"?>
```

XML Top-level node. The XML declaration node is followed by an XMI declaration node, stating that everything inside this node is XMI, and not any other XML information. The whole model consists of subordinate nodes located inside the XMI declaration node.

XMI top-level node

```
1 <xmi:XMI xmi:version="20131001" . . . >
2 . . . the proper model-related contents comes here . . .
3 </xmi:XMI>
```

Namespace declarations. The inner XML nodes containing model information represent all language element instances[4] used in the model. The language elements come from the proper UML or from the SysML profile or from the so-called StandardProfile[5] and, potentially, further profiles. We will see in the next paragraph, that every language element *instance* used in the model is represented as an XML node having

XML namespaces

- a general element classification as node name,
- the language element type as XMI type name, and
- either `uml` or the profile name as the name of the node namespace.

The namespaces used (i.e. `uml` and the profile names) are introduced in the `xmi:XMI` top-level node introduced above and specified like

```
1 <xmi:XMI xmi:version="20131001"
2 xmlns:xmi="http://www.omg.org/spec/XMI/20131001"
3 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
4 xmlns:Blocks="http://www.eclipse.org/papyrus/sysml/1.6/SysML/Blocks"
```

[4] instances of Blocks, requirements, state machines, associations, ...

[5] see `https://www.omg.org/spec/UML/About-UML/`

```
5 xmlns:PortsAndFlows="http://www.eclipse.org/papyrus/sysml/1.6/SysML/
    PortsAndFlows" xmlns:Requirements="http://www.eclipse.org/papyrus/sysml
    /1.6/SysML/Requirements"
6 xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
7 xmlns:standard="http://www.eclipse.org/uml2/5.0.0/UML/Profile/Standard"
8 xmlns:uml="http://www.eclipse.org/uml2/5.0.0/UML"
9 xsi:schemaLocation="http://www.eclipse.org/papyrus/sysml/1.6/SysML/Blocks
    http://www.eclipse.org/papyrus/sysml/1.6/SysML#//blocks http://www.eclipse
    .org/papyrus/sysml/1.6/SysML/PortsAndFlows http://www.eclipse.org/papyrus/
    sysml/1.6/SysML#//portsandflows http://www.eclipse.org/papyrus/sysml/1.6/
    SysML/Requirements http://www.eclipse.org/papyrus/sysml/1.6/SysML#//
    requirements">
```

In this example taken from our turn indication model, each entry like `xmlns:Blocks` introduces a new namespace: keyword `xmlns` stands for **XML namespace**, and the namespace name to be introduced is given after the colon. The example above introduces the XML namespaces `xmi, xsi, Blocks, PortsAndFlows, Requirements, ecore, standard, uml`. As we can see from this example, the whole SysML is structured into several namespaces, whereas all UML language elements belong to the same (very large) namespace `uml`.

The UML model section. The whole model is structured into

- the proper UML model section containing only language element instances of the UML itself, and
- the profile-specific language element instances appended to UML part.

The UML model part is encapsulated in an XMI node like

```
1 <uml:Model xmi:id="_bS-eEIL_EeqQ7ILEbQ0thw" name="TurnIndicationJP">
2 . . . the UML model part without profile-specific elements . . .
3 </uml:Model>
```

You see that `Model` is a UML language element[6] which has a name (in this example, it's `TurnIndicationJP`) and a unique identification which is marked by `xmi:id` and uniquely identifies each model element in the given XMI file. The model node is terminated by `</uml:Model>`.

The XMI node entries like `xmi:id` and `name` are called **node attributes**[7], and the concept for adding information to nodes by means of these attributes comes from the XML. We will, however, talk only about XMI nodes in the remainder of these lecture notes, since – apart from the first node stating

[6] In fact, it's a special type of UML `Package`.

[7] not to be confused with the attributes of a class that we will introduce below

that the file is an XML document – there are only UML-specific and profile-specific XMI nodes to consider.

Before going into the details what is inside the model node, we will first discuss the profile-specific model elements appended to the model node. This is because some UML language element instances *inside* the model node are only there since they are needed to represent the detailed information about the profile language element instance.

Profile-specific model elements. The profile-specific model elements are appended to the model node in arbitrary order. As first example, we consider a language element instance of a block. For example, the block representing the RearController is specified by the XMI node

Blocks

```
1 <Blocks:Block xmi:id="_qbUxwJNcEeqNGJG7YISZag"
2               base_Class="_qbS8kJNcEeqNGJG7YISZag"/>
```

Oops, there's not much information provided: we just see that the model element is a `Block` and has an `id`. But, as explained above, a block is nothing but a UML class. Therefore, we can expect that inside the UML-specific model part, there is some `Class` instance containing all the details of the block under consideration. And this is of course the case: the block attribute `base_Class` refers to a class instance in the UML model by means of the unique model element identification `_qbS8kJNcEeqNGJG7YISZag`.

UML model elements in the model section. Under this identification, we find a `Class` instance in the UML mode section specified as shown in Listing 2.1. To decode the type, first note that the node name is `packagedElement`: no namespace indication, no mentioning about classes.

- The namespace `uml` is not mentioned for nodes *inside* the `uml:Model` node.
- The node name '`packagedElement`' is an attribute of the UML model element `Package`, because the UML class we wish to investigate resides inside a package. Every model element inside this package is represented by a `packagedElement` node inside the package node. Model elements that may be placed into a package must be of type `PackageableElement`. This is a very general class of language elements, and the UML `Class` element is of course a sub-class of `PackageableElement`.

Listing 2.1: UML class information for SysML block RearConttroller.

```
<packagedElement xmi:type="uml:Class" xmi:id="_qbS8kJNcEeqNGJG7YISZag" name=
  "RearController">
        <ownedAttribute xmi:type="uml:Port" xmi:id="_JlGDIJN-EeqNGJG7YISZag"
            name="LeverPositionIN" type="_TbLCoI03Eeq_BtA7y9KmTw"
            aggregation="composite" isConjugated="true"/>
        <ownedAttribute xmi:type="uml:Port" xmi:id="_WR8wQJN-EeqNGJG7YISZag"
            name="EmerSwitchPressedIN"
            aggregation="composite" isConjugated="true">
          <type xmi:type="uml:DataType"
            href="pathmap://SysML16_LIBRARIES/..._Boolean"/>
        </ownedAttribute>
        . . . further ports . . .
        <ownedAttribute xmi:type="uml:Property" xmi:id="_gA_EEJ6EEeqfg"
          name="candriver"
          type="_S1814J0REeqVNdKVvHl0_A" aggregation="shared"
          association="_gA5kgJ6EEeqfg-0S5eTmFw"/>
        <ownedAttribute xmi:type="uml:Property" xmi:id="_jYl5Ep6EEeqfg"
          name="controllogic"
          type="_Wf6mgJ0QEeqVNdKVvHl0_A" aggregation="shared"
          association="_jYlSAJ6EEeqfg-0S5eTmFw"/>
        <ownedAttribute xmi:type="uml:Property" xmi:id="_mtXcg56EEe"
          name="lampCtrlSlaveR"
          type="_wQXSsJhjEeqIY5MUq7ZxfQ" aggregation="shared"
          association="_mtXcgJ6EEeqfg-0S5eTmFw"/>
        <ownedAttribute xmi:type="uml:Property" xmi:id="_xNqyg56EEeq"
          name="lampCtrlSlaveL"
          type="_wQXSsJhjEeqIY5MUq7ZxfQ" aggregation="shared"
          association="_xNqygJ6EEeqfg-0S5eTmFw"/>
        <ownedAttribute xmi:type="uml:Property" xmi:id="_4IyxkJ6EE"
          name="pinProgR"
          type="_5XibsJjpEeqpEYL2Okl47g" aggregation="shared"
          association="_4IyKgJ6EEeqfg-0S5eTmFw"/>
        <ownedAttribute xmi:type="uml:Property" xmi:id="_93Tzkp6EEe"
          name="pinProgL"
          type="_5XibsJjpEeqpEYL2Okl47g" aggregation="shared"
          association="_93TMgJ6EEeqfg-0S5eTmFw"/>
        <ownedAttribute xmi:type="uml:Property" xmi:id="_EAXcg56FEeqfg"
          name="pwrRightLamp"
          type="_E5MCwJhiEeqIY5MUq7ZxfQ" aggregation="shared"
          association="_EAXcgJ6FEeqfg-0S5eTmFw"/>
        <ownedAttribute xmi:type="uml:Property" xmi:id="_JBIAo56FEeqfg"
          name="pwrLeftLamp"
          type="_E5MCwJhiEeqIY5MUq7ZxfQ" aggregation="shared"
          association="_JBIAoJ6FEeqfg-0S5eTmFw"/>
</packagedElement>
```

You can see in Listing 2.1, that the rear controller's concrete type `Class` has been specified in the XMI node attribute `xmi:type` which is always used to specify the concrete type of language elements which are only described as a `PackageableElement` instances in the node name. As a class,

the rear controller has attributes that are wrapped into XMI nodes named `ownedAttribute`. Each attribute is an instance of a language element type, and the latter is specified in the `xmi:type` node attribute. Since types may come from UML, SysML or other profiles, the type specification is associated with a name space. For example, the first attribute is of UML type port, and the port is named `LeverPositionIN`. Now we remember that ports are typed, and therefore, a **type reference** is found under node attribute `type`. Please note the subtle difference:

Port attribute

- The language element type (in this example, a UML port) is specified in node attribute `xmi:type`, whereas
- the type of the specific language element instance (in this example, a UML port instance) is specified in the `type` attribute, that is, *without the* `xmi` *prefix*.

The type is referenced again by an `xmi:id` (`_TbLCoI03Eeq_BtA7y9KmTw` in our case), so we look up the type definition in the UML model part. Indeed, we find a node with this id and the following information specified in Listing 2.2. It is easy to decipher that `LeverPositionIN` is a UML enumeration type which has enumeration literals `NEUTRAL`, `LEFT`, and `RIGHT`. The `specification` XMI node associates an integer constant with each enumeration literal, to look this up, we have to follow another `xmi:id` reference.

Listing 2.2: UML specification of enumeration type LeverPosition.

```
<packagedElement xmi:type="uml:Enumeration" xmi:id="_TbLCoI03Eeq_BtA7y9KmTw"
                 name="LeverPosition">
    <ownedLiteral xmi:type="uml:EnumerationLiteral"
      xmi:id="_byq5EI03Eeq_BtA7y9KmTw" name="NEUTRAL">
      <specification xmi:type="uml:LiteralInteger" xmi:id="..."/>
    </ownedLiteral>
    <ownedLiteral xmi:type="uml:EnumerationLiteral"
      xmi:id="_f_8KwI03Eeq_BtA7y9KmTw" name="LEFT">
      <specification xmi:type="uml:LiteralInteger" xmi:id="..."/>
    </ownedLiteral>
    <ownedLiteral xmi:type="uml:EnumerationLiteral"
      xmi:id="_iwwskI03Eeq_BtA7y9KmTw" name="RIGHT">
      <specification xmi:type="uml:LiteralInteger" xmi:id="..."/>
    </ownedLiteral>
</packagedElement>
```

Back and forth between UML model and profile part This example of the port attribute for a class has already shown the basic mechanisms of XMI files: model elements are identified by XMI node names and additional type information, and further associated information is retrieved via `xmi:id` references. There is, however, one more problem to keep in mind: the UML model elements found in the UML model part of the XMI file, may again be referenced by other profile element instances. For example, the port attribute associated with class RearController may be a *full port* in SysML, and therefore also referenced by the profile section attached to the model section in the XMI file. Indeed, this is the case for the port `LeverPositionIN` discussed above: In the profile section of the XMI file, we find an XMI node

Further references from the profile part

```
<PortsAndFlows:FullPort xmi:id="_JlH4UJN-EeqNGJG7YISZag"
                        base_Port="_JlGDIJN-EeqNGJG7YISZag"/>
```

which is a `FullPort` and references our UML port `LeverPositionIN` in its `base_Port` attribute. This is a bit cumbersome: When tracing a SysML block back to a UML class and analysing the class attributes, then we have to check whether the class attributes are themselves referenced from other profile-specific model elements. For example, the class RearController discussed above also has a part attribute `candriver`. This is typed by the UML class `CANdriver` which can be found in the UML model section under id `_Sl814J0REeqVNdKVvHl0_A`. Class `CANdriver`, however, is associated with a SysML block specified in the profile section of the XMI file:

```
<Blocks:Block xmi:id="_Sl-rEJ0REeqVNdKVvHl0_A"
              base_Class="_Sl814J0REeqVNdKVvHl0_A"/>
```

Unfortunately, we have to accept the XMI file structure as it is, so we cannot do anything about being forced to continuously go back and forth between the UML-specific and the profile-specific part. We note, however, that for the evaluation of XMI files in a program, a map from `xmi:id` to XMI nodes will be extremely helpful.

Chapter 3

The Libxml2 Library

The Libxml2 library[1] has been developed for parsing and changing XML documents from C or C++ programs. It is a very well-established library which has been quite stable for years. The basic idea of the library is to create an abstract syntax tree (ADT) from an XML document and provide ADT traversal operations, operations for reading XML node content, and for changing node content.

3.1 Installation

We do not need the source code of the Libxml2 library, precompiled binaries plus header files are ok. Installation details can be found in `http://xmlsoft.org/downloads.html` for all platforms.

3.2 Documentation

Though we will introduce the most important Libxml2 functions here in the lecture notes, it is advisable to look up the original documentation every now and then. We will mostly use the so-called **Tree-API** which is documented in `http://xmlsoft.org/html/libxml-tree.html`.

[1] `http://xmlsoft.org`

3.3 Creating Executables Using Libxml2

C-programs using the Libxml2 API need to include two files:

```
1 #include <libxml/xmlmemory.h>
2 #include <libxml/parser.h>
```

If you are working with a software development IDE, you should be able to configure your project in such a way that Libxml2 is always linked to your software code, and that the header files are available in the include path. The initial code for the model checker to be completed in the exercise below consists of the files

```
XmiUtils.c
XmiUtils.h
checker.c
checker.h
list.h
main.c
```

On my Mac OSX platform, the executable can be generated by the command

```
cc -o xmichecker \
      -I./ -I/usr/local/opt/libxml2/include/libxml2 \
      *.c -lxml2
```

- `cc` is the C-compiler invocation command. On my platform, it is linked to `clang`, on Linux, it will typically be linked to `gcc`, or you have to give the `gcc` command instead of `cc`.
- The `-o xmichecker` option says that the executable program should be named `xmichecker`.
- `-I./` tells the compiler to look for include files in the local directory (where the files listed above are located and where you give the `cc` command).
- The `-I/usr/local/opt/libxml2/include/libxml2` tells the compiler to look also in this directory for further include files. This is the location on my computer where the Libxml2 header files are located. On your

computer, a prefix of this path needs to be exchanged, depending on where your library header files have been installed.

Please observe that the Libxml2 include files are always referenced with root directory `libxml`, so your include path must be structured in a way that the header file can be found in the path's sub-directory `libxml`. For example, header file `parser.h` resides in

```
/usr/local/opt/libxml2/include/libxml2/libxml
```

on my computer. Therefore, I use the path prefix

```
/usr/local/opt/libxml2/include/libxml2
```

in the `-I` include option.

- `*.c` tells the compiler to compile all C-files in the working directory.
- `-lxml2` tells the linker to bind the Libxml2 to the other object files when creating the executable.

On Windows platforms, the compile and link command looks different, but I expect that you will use an IDE anyway. Note, however, that you can also compile and link your programs using explicit command in the Windows command shell.

Chapter 4

Static SysML Model Checking

Using Libxml2, we can build simple checkers for the static model semantics. To this end, a coding framework has been provided with this session in archive file `XMI-tools.zip`. This framework contains the files

```
XmiUtils.c
XmiUtils.h
checker.c
checker.h
list.h
main.c
```

mentioned above.

Main program main.c This file contains the main function of the program. It opens the XMI file using an auxiliary function from `XmiUtils.c`, creates the mapping from `xmi:id` to Libxml2 nodes of the AST, and calls the model checking functions one by one. As a first exercise, you are asked to program the checking algorithm for requirements, as specified in Exercise 5.1 in checking function `checkRequirements()`.

Checker library checker.c This file contains all the checking functions for the static model semantics. At present, it only contains the code frame for function `checkRequirements()`.

Utility library XmiUtils.c This library has been created to facilitate basic search functions for model checking. The library function interfaces are specified in `XmiUtils.h`, but some basic explanations are given here in the lecture notes.

`openModel()` opens the specified file, checks whether it is a true XMI document, and returns a pointer to the AST created by the Libxml2 while parsing the document.

`setupIdMap()` traverses the complete AST and creates a list mapping xmi:id to the corresponding nodes in the AST.

`findNodeById()` returns a pointer to the AST node associated with a given xmi:id. This function is needed in many different situation. We have discussed, for example, how to find the class associated with a given SysML requirement by looking up the XMI node with xmi:id specified in the requirement's attribute `base_Class`.

`findNodesByNsNodeName()` This is the first library function having a *list* of nodes as search result. Given a namespace name and an XMI node name (like `Block` or `nestedClassifier`), the function collects all matching AST nodes in a list which is internally created and managed inside the XmiUtils library. Users provide a pointer to a *list handle* which will be written to by this function when the list has been created and is used for list identification for all subsequent operations on the list. Moreover, the function returns an iterator that can be used to retrieve the list elements one by one.

`nextSearchResult()` This function takes the list handle and an iterator as input and returns a pointer to the next AST node retrieved from the list. If the list has been completely traversed, NULL is returned.

`newIterator()` If a search result list hast to be traversed several times, for example, in nested loops, additional iterators can be created by means of function `newIterator`.

The code frame for function `checkAtomicRequirements()` shows how these library functions are applied.

List handling header file list.h The Linux kernel has a very elegant concept for generic list handling in the C programming language. The associated macros are contained in `list.h` and are re-used here. I have encapsulated most of the list handling in the list access functions contained in `XmiUtils.c`. However, if you are interested in how generic list handling works with the C programming language, please consult Appendix A. This knowledge is also required if you wish to understand the details of the XmiUtils library and desire to make additions to this library.

Chapter 5

Questions and Exercises

5.1 Check Atomic and Non-Atomic Requirements

Please extend the sample code of the xmlchecker handed out with these lecture notes and complete function `checkAtomicRequirements()` by programming the following checks:

1. Every requirement in the model which has been marked as *atomic* by setting the class attribute `isLeaf` to text string `true`, is free of any nested classifiers.

2. Every requirement in the model which has been marked as *non-atomic* by setting the class attribute `isLeaf` to text string `false`, or by leaving out this attribute[1], has at least one nested classifier which is a requirement.

If an erroneous requirement has been detected, print out

- Requirements id,
- Requirements text,
- Error message

to the console (just use `printf` for this).

[1]The attribute has default value `false`, so it's not necessary to state this explicitly.

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] Object Management Group. XML Metadata Interchange (XMI) Specification, Version 2.5.1. Technical report, Object Management Group, 2015. `http://www.omg.org/spec/XMI/2.5.1`.

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

[4] Object Management Group. OMG Systems Modeling Language (OMG SysML), Version 1.6. Technical report, Object Management Group, 2019. `http://www.omg.org/spec/SysML/1.4`.

Appendix A

List Handling

The Linux list package re-used here for the XMI-tools is designed as follows.

As a first step, a doubly-linked ring list, whose elements do not contain any user data is introduced using the following structure.

The list_head structure

```
1 struct list_head {
2     struct list_head* next;
3     struct list_head* prev;
4 };
```

An abstract list data type (still without user data) is built with this structure as follows.

List handling macros

1. An empty list `myList` is created by defining a list head and initialising it with a macro from `list.h`:

   ```
   struct list_head myList = LIST_HEAD_INIT(myList);
   ```

   The initialisation macro lets the `next` pointer and the `prev` pointer point to list head itself, so empty lists are represented by list heads whose predecessors and successors are again the list head.

2. New list elements are also created by allocating new instances of `struct list_head`, typically using dynamic memory allocation. The new elements (let's say that `elemPtr` points to the allocated instance) are inserted at the end of the list using the macro call `list_add(&myList,elemPtr);`. To this end, the macro performs the following asssignments.

   ```
   1 elemPtr->next = &myList;
   2     // Let successor of elem point to list head
   3 elemPtr->prev = myList.prev;
   ```

```
    // Let predecessor of elem point to the old last element
myList.prev->next = elemPtr;
    // The old last element gets elem as its successor
myList.prev = elemPtr;
    // The list head's predecessor points to elem
```

The effect of these assignments is that `*elemPtr` is the new last element of the list.

3. To traverse a list, recall that the list head does not represent a list element, but is used to present the empty list by pointing to itself. Therefore, list traversal starts at the successor of the list head (in our example here, at `struct list_head* p = myList.next` and stops when the actual element pointer `p` points to the list head. As a consequence, list traversal is realised by the following loop.

```
struct list_head* p;
for ( p = myList.next; p != &myList; p = p->next ) {
...
}
```

Traversal from last element to first element is performed by a loop of the form

```
for ( p = myList.prev; p != &myList; p = p->prev ) {
...
}
```

4. When deleting elements during list traversal, it must be ensured that the pointer to the current element is updated accordingly. To this end, the macro `list_rm_loopsafe()` is used:

```
struct list_head* p;
for ( p = myList.next; p != &myList; p = p->next ) {
   if ( ... ) {
      list_rm_loopsafe(p);
   }
}
```

Create lists with user data

For creating lists with user data, a structure containg the user data *and* a component of type `struct list_head` is created. For example, the meta classes of the UML are represented in a structure (see file `utils.h`)

```
typedef struct ClassSpec {
    /** Element id specified for type in XMI file */
    char* id;
    char* pathPrefix;
```

```
5     char* packagename;
6     char* classname;
7     ...
8     /** List of ClassSpec_t entries */
9     struct list_head h;
10 } ClassSpec_t;
```

The start of the list is again represented by a variable of type `struct list_head`. For example, the list of all UML meta classes has list head definition and initialisation

```
1 struct list_head allClassesHead=LIST_HEAD_INIT(allClassesHead);
```

Now suppose that `ClassSpec_t* cls` is a pointer to a user data instance specifying a new meta class. This instance is inserted into the list of meta classes by using the `list_add()` macro introduced above as follows:

```
1 list_add(&allClassesHead,&cls->h);
```

Retrieve user data during list traversal

When traversing a list with user data, this is again performed by traversing the `struct list_head` instances. To access the user data of the list element, however, the pointer to the start of the element has to be reconstructed from the pointer to list head element. This is done by means of the `container_of()` macro which in turn uses the `offsetof()` macro specified as follows:

```
1 /**
2  * The following auxiliary macro returns the address offset of a
3  * structure component within the structure. It is evaluated during
4  * compilation time, never at runtime, so NULL-pointer dereferentiation
5  * is "harmless". Some compilers have this macro as built-in function.
6 */
7 #define offsetof(TYPE, MEMBER) ((unsigned int) &((TYPE *)0)->MEMBER)
8
9 /**
10 * container_of - cast a member of a structure out to the
11 * containing structure
12 * @param ptr: the pointer to the member.
13 * @param type: the type of the container struct this
14 *         is embedded in.
15 * @param member: the name of the member within the struct.
16 *
17 */
18
19 #define container_of(ptr, type, member) \
20 ( (type *)( (char *)ptr - offsetof(type,member) ) )
```

With these macros at hand, list traversal with user data is performed as follows. Again, we use the list of all UML meta classes as an example.

```
struct list_head* lh;
for ( lh = allClassesHead.next;
      lh != &allClassesHead;
      lh = lh->next ) {
  // Get pointer to start of user data
  // which has type ClassSpec_t
  ClassSpec_t* cls = container_of(lh, ClassSpec_t, h);
  ... process cls ...
}
```