Master Thesis

Sandrina Backhauß

Code Generation for UML Activity Diagrams in Real-Time Systems

October 4, 2016

supervised by:
Prof. Dr. Sibylle Schupp (First Examiner)
Prof. Dr. Ralf God (Second Examiner)
Detlef Götting (Airbus)

Hamburg University of Technology (TUHH)
Technische Universität Hamburg-Harburg
Institute for Software Systems
21073 Hamburg
Statutory Declaration

I, Sandrina Backhauß, declare that I have authored this thesis independently, that I have not used sources/resources other than the declared, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources. Neither this thesis nor any other similar work has been previously submitted to any examination board.

Hamburg, October 4, 2016
Abstract

In this thesis, a code generator is developed, which transforms UML activity diagrams into ANSI-C source code. The source code should be used in real time systems in the context of avionic systems and thus has to fulfil accreditation rules and safety issues. An algorithm that bridges the semantic gap between the UML and the C language is designed and implemented successfully. To achieve this, algorithms from the field of graph theory, stereotypes and function calls into a real-time operating system are used. Termination considerations are made, and unit and integration tests are performed to validate functional correctness and robustness of the algorithm.
# Contents

<table>
<thead>
<tr>
<th>Acronyms</th>
<th>xi</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1. Scope</td>
<td>1</td>
</tr>
<tr>
<td>1.2. Outline</td>
<td>2</td>
</tr>
<tr>
<td>2. Background</td>
<td>5</td>
</tr>
<tr>
<td>2.1. Literature Review</td>
<td>5</td>
</tr>
<tr>
<td>2.2. Input Format</td>
<td>7</td>
</tr>
<tr>
<td>2.2.1. Environment</td>
<td>8</td>
</tr>
<tr>
<td>2.2.2. Unified Modelling Language (UML) Elements</td>
<td>9</td>
</tr>
<tr>
<td>2.2.3. XML Metadata Interchange (XMI)</td>
<td>15</td>
</tr>
<tr>
<td>2.3. Output Format</td>
<td>16</td>
</tr>
<tr>
<td>2.4. Transformation</td>
<td>17</td>
</tr>
<tr>
<td>3. Code Generation</td>
<td>19</td>
</tr>
<tr>
<td>3.1. XMI Parser</td>
<td>19</td>
</tr>
<tr>
<td>3.2. Internal Structure</td>
<td>21</td>
</tr>
<tr>
<td>3.3. Algorithm</td>
<td>23</td>
</tr>
<tr>
<td>3.4. Back-End</td>
<td>34</td>
</tr>
<tr>
<td>3.5. Example: Greatest Common Divisor</td>
<td>35</td>
</tr>
<tr>
<td>4. Evaluation and Validation</td>
<td>39</td>
</tr>
<tr>
<td>4.1. Algorithm Termination</td>
<td>39</td>
</tr>
<tr>
<td>4.2. Algorithm Correctness</td>
<td>39</td>
</tr>
<tr>
<td>4.2.1. Unit Testing</td>
<td>41</td>
</tr>
<tr>
<td>4.2.2. Integration Testing</td>
<td>44</td>
</tr>
<tr>
<td>4.2.3. Static Code Analysis</td>
<td>45</td>
</tr>
<tr>
<td>4.3. Evaluation</td>
<td>46</td>
</tr>
<tr>
<td>5. Conclusion</td>
<td>49</td>
</tr>
<tr>
<td>5.1. Outlook</td>
<td>49</td>
</tr>
<tr>
<td>A. Transformation Rules</td>
<td>51</td>
</tr>
<tr>
<td>B. Greatest Common Divisor (gcd)</td>
<td>55</td>
</tr>
<tr>
<td>C. Test Cases</td>
<td>65</td>
</tr>
<tr>
<td>D. Test Coverage</td>
<td>73</td>
</tr>
<tr>
<td>D.1. Assemblies</td>
<td>73</td>
</tr>
<tr>
<td>D.2. ActivityGen.Algorithm</td>
<td>76</td>
</tr>
<tr>
<td>D.2.1. Summary</td>
<td>76</td>
</tr>
</tbody>
</table>
List of Figures

1.1. V-model (Detlef Götting) ............................................. 2
2.1. SysML Diagram Types [19] ........................................ 8
2.2. (a) UML initial node, (b) UML activity final node, (c) UML flow final node 9
2.3. (a) UML decision node, (b) UML merge node, (c) UML fork node (d) UML join node ...................................................... 10
2.4. UML opaque action node with input and output pin, UML call behaviour action node, UML send signal action node, UML accept event action node 12
2.5. UML activity parameter node ....................................... 12
2.6. UML data store node, UML central buffer node ................... 13
2.7. UML activity partition node ......................................... 13
2.8. UML sequence structured activity node ............................. 14
2.9. UML expansion region node ......................................... 14
2.10. Software Development Process (Detlef Götting) ................ 18

3.1. workflow ............................................................... 19
3.3. UML activity class ..................................................... 20
3.4. Graph class ........................................................... 22
3.5. A fork expands the main task by the parallel tasks Task1 and Task2. The join node joins these tasks back to one. The blue edges show the flow the current task takes. ................................................. 24
3.6. (a) A valid graph, which can be sorted in topological order, (b) a graph with cyclic dependencies, which cannot be sorted in topological order. ... 27
3.7. A branch, which will be transformed into an if-then-else, a branch, which will be transformed into a switch-case and a loop, which will be transformed into a while-loop ................................................. 32
3.8. Overview of the internal classes for activity diagrams ............ 37

B.1. gcd internal graph .................................................... 55
B.2. gcd internal graph after createCLocalVariables ................. 56
B.3. gcd internal graph after transformAction2C ....................... 57
B.4. gcd internal graph after applying the depth first search .......... 58
B.5. gcd internal graph after mergeBranches in iteration 1 .......... 59
B.6. gcd internal graph after mergeLoops in iteration 1 ............... 60
B.7. gcd internal graph after mergeCNodes in iteration 2 ............. 61
B.8. gcd internal graph after mergeBranches in iteration 2 .......... 62
B.9. gcd internal graph after mergeCNodes in iteration 3 ............. 63

E.1. Gcd: Model main function ......................................... 77
E.2. Gcd: Internal graph (control flow edges) main function ........ 77
E.3. Gcd: Model gcd function ........................................... 78
E.4. Gcd: Internal graph (control flow edges) gcd function .......... 78
List of Figures

E.5. Calculator: Model main function .............................................. 79
E.6. Calculator: Internal graph (control flow edges) main function .......... 79
E.7. Calculator: Model calculator function ....................................... 80
E.8. Calculator: Internal graph (control flow edges) calculator function .... 80
E.9. Calculator: Model operator function ......................................... 81
E.10. Calculator: Internal graph (control flow edges) operator function ..... 81
E.11. Iteration 3: Model structureNode function ................................. 82
E.12. Iteration 3: Internal graph (control flow edges) structureNode function . 82
Acronyms

CSP  Communicating Sequential Process
GCD  Greatest Common Divisor
GUI  Graphical User Interface
MBE  Model-Based Engineering
MISRA Motor Industry Software Reliability Association
OAL  Operation System Abstraction Layer
OMG  Object Management Group
PTC  Parametric Technology Corporation
RTOS Real-Time Operating System
SCS  Software Coding Standard
SLDL System Level Design Language
SysML System Modeling Language
UML  Unified Modelling Language
XML Extensible Markup Language
XMI  XML Metadata Interchange
1. Introduction

Model-Based Engineering (MBE) is a future driven topic at Airbus Operations GmbH. It should replace the requirement-based approach, which is used at the moment. In MBE, a system is described graphically in a modelling language such as the Unified Modelling Language (UML) instead of a textual representation. This change leads to a more comprehensible representation, and misunderstandings between the customer and the designer can be discovered early in the development life cycle. Currently, the development life cycle follows the V-model. In the V-model as depicted in figure 1.1, the specification is divided into several steps. With each step, the specification becomes more detailed. General requirements for the whole system are placed at the top level whereas the specification of small units takes place at the bottom. After the system is implemented, different testing steps and integration are required. The test cycle begins with unit tests and ends with acceptance tests. Each step of testing maps to a corresponding step in the specification. In this thesis, the steps of the V-model should be reduced through automated code generation and thus the time-to-market will be reduced as well. The UML activity diagram type can model individual units of a system. After the creation of a model, the activity diagram can be automatically transformed into source code. In future work, the transformation presented in this thesis will be validated according to the DO-330\textsuperscript{1} tool qualification. After qualifying the code generator, it will be possible to verify the specification against the model instead of the source code. Thus, the V-model is transformed into a Y-model \cite{9}, where the edge from Item Implementation to Executable Item is relocated so that a connection between Item Requirements and Executable Item is established. As a result, errors can be identified earlier, and the earlier an error is found in the development life cycle, the lower the costs will be of fixing it.

1.1. Scope

This thesis focuses on the transformation of UML activity diagrams into ANSI-C\textsuperscript{99} source code. The first step to developing such a transformation is to identify the relevant UML elements that can belong to a UML activity diagram. Additionally, the ANSI-C constructs used in the context of avionic real-time systems have to be determined. The real-time context adds additional requirements to the transformation since the calls to operating system functions dealing with real-time have to be considered. In particular, to compile ANSI-C files for the use in a real-time system, the Operation System Abstraction Layer (OAL) functions, developed internally by Airbus, need to be integrated. With the use of the OAL, the source code becomes independent of the specific Real-Time Operating System (RTOS). After the identification of involved items, the UML input elements have to be mapped to the output ANSI-C constructs. Following this preparation, the mapping has to be implemented as an automatic program to

\textsuperscript{1}DO-330 is an avionic standard which contains the process for tool qualification.
generate ANSI-C source code. This source code has to satisfy accreditation rules and safety issues. Furthermore, it has to comply with design, coding and quality standards. Finally, the resulting software needs to be tested.

As described in 2.1 no commercial code generator exist that fulfils all these requirements. Thus, a novel code generator for UML activity diagrams is implemented and presented in this thesis. The mappings and transformations from the UML to ANSI-C are the core achievement of this thesis and use different sets of rules and algorithms from the field of graph theory.

1.2. Outline

This thesis is divided into a background chapter, an implementation chapter and an evaluation and validation chapter. The background is described in chapter 2. This chapter is structured in four sections. At first, section 2.1 refers to the related work and commercial code generators. Following this, the input format required for the transformation will be defined in section 2.2. Therefore, the environment especially the Parametric Technology Corporation (PTC) Integrity Modeler, which is used to design the models, is presented in subsection 2.2.1. Additionally, the description and the graphical notations of the UML elements according to the literature are noted in subsection 2.2.2. Finally, the XML Metadata Interchange (XMI) format is introduced in subsection 2.2.3. After the input format is specified, the output format is defined in section 2.3 with respect to the context of avionic real-time systems and ANSI-C as target language. To complete this chapter, the integration of the automatic and algorithmic code generation into the development process and the challenges are described in section 2.4.

In chapter 3 the implementation of a code generator for activity diagrams is presented. Firstly, it is shown how the XMI parser is constructed in section 3.1. The parser trans-
forms the XMI format into an internal structure, which is presented in section 3.2. Then an algorithm, which operates on the internal structure, is developed and explained step by step in section 3.3. The algorithm is responsible for a correct mapping of the input format into the output format. The output of the generated ANSI-C source code into a file happens in a separate step, which is presented in section 3.4. Finally, the algorithm is explained with an example model of the Greatest Common Divisor (GCD) in section 3.5.

The evaluation and validation of the code generator implementation from chapter 3 is described in chapter 4. First of all, section 4.1 shows that the developed algorithm terminates. Additionally, it is shown that the algorithm works correctly in section 4.2. For this, the mapping is validated by unit tests in subsection 4.2.1 and integration tests in subsection 4.2.2. In subsection 4.2.3, static code analysis checks the generated source code file for compliance with the output format. To complete this chapter, the challenges described in section 2.4 are evaluated in section 4.3. Finally, chapter 5 gives a conclusion and an outlook.
2. Background

In the following chapter, the prerequisites of this thesis will be presented. The chapter describes the different approaches to code generation and their implementation in related work. Then the input format and the environment used in this thesis will be explained. Afterwards, the expected output format will be introduced. Finally, the tasks of a code generator will be represented.

2.1. Literature Review

This section will introduce the basic approaches to code generation in the literature. The transformation between different languages can be performed on a step by step basis using rules or code patterns, or it can be realised by means of a context-free grammar.

System Level Design Languages (SLDLs) can be automatically transformed into ANSI-C source code for an RTOS [30], [7]. Yu, Dömer and Gajski implement a code generator for SpecC as an example of an SLDL [30]. During the transformation that they apply, intermediate models are generated. In several stages "the system specification is gradually refined from an abstract idea down to an actual implementation" ([30, p.464]). In one step, the source code generation works on an intermediate model that consists of several processing elements. Each processing element consists of multiple tasks. These processing elements and tasks are constructed in an earlier refinement step. During code generation, ANSI-C source code replaces the SLDL description of each task successively. The conversion is based on multiple rules, and each of these rules represents a mapping from an SLDL element to a corresponding ANSI-C language element. The rules have been used to develop an algorithm which takes an SLDL description and transforms it into ANSI-C source code automatically. This algorithm neither performs task scheduling nor signalling but instead hands down the responsibility for this to the RTOS. After the transformation to C finishes, the generated files can be compiled to binary code to run on a target system.

Ranta proposes a similar approach to compile high-level programs to binary code [24]. At first, a general mapping needs to be defined to bridge the semantic gap between source and target language. Due to the semantic gap, it is possible for multiple high-level code elements to map to a single binary code element. In the same way, one high-level code element can be translated into multiple machine code statements. Interference rules can be used to construct the code generator. But due to the potential complexity of such rules for the whole code generator, the authors propose to use a pseudocode notation instead. Because of the mentioned semantic gap, some information needs to be ignored during compilation. Other information from the high-level code may have global implications, and thus the transformation can not always be solved locally by element-wise translation. To keep this extra information which might be data on a variable's address and or function's type, it can be stored in the compilation environment.

Gessenharter discusses a Java code generation for UML associations[10]. The authors'
algorithm is designed to extend existing code generators with the ability to translate UML associations correctly using alternative code patterns. To achieve this task, the required UML association elements are determined, and the concept of an association is presented. Following this, the new method to translate UML associations is compared to other code generators currently available. Due to the incompleteness of the available code generation patterns, an extension is proposed that attempts to solve existing problems and adds missing functionality. Some problems unsolved by this paper [10] are addressed by the same authors in their follow-up work [11]. The authors’ solution is to extend the input UML model by inserting additional classes into it. These new class-instances implement the singleton pattern and represent the association between two existing classes. This avoids that "an instance discontinues referencing a tuple that the associated instance is still pointing at" [11, p.19].

Another approach is presented by Chanda et al. [2]. The authors describe how UML diagrams can be verified automatically regarding syntax and semantics. As a prerequisite, the UML model needs to be converted into the XMI format. Afterwards, the XMI file has to be treated as a single string that conforms to the context-free grammar proposed in the paper. The grammar is presented only for UML class and UML sequence diagrams and allows a formalisation of the UML model. "The production rules, terminals, non-terminals are chosen and proposed such that it adheres to the notations of UML 2.0 standard." [2]. Thus, verification criteria like rules for syntactic correctness and consistency, which are defined in the paper, can be applied. This approach could be extended so that the grammar can be used to transform the string to ANSI-C code. The verification rules could be used to model the pre- and postconditions.

An additional approach to formalise UML activity diagrams is proposed by Xu et al. [29]. The semantic gap inhibits formal system behaviour analysis, but by means of Communicating Sequential Processes (CSPs) the behaviour can be formalised and checked afterwards. The paper provides a mapping from UML activity diagram elements to the formal language CSP. Each UML element is transformed separately based on a function defined by the authors.

The authors Audsley et al. state that even though code generation is not a new concept, existing code generators lack the ability to "enforce any specific coding standards or structured techniques" [1]. In addition, the produced source code sometimes is unreadable and of uncertain quality. Existing code generators focus on narrow fields of applications and therefore do not cover a wide variety of models and their elements. The two code generators that have been assessed as a reference for this thesis, one from PTC and one from Sinelabore RT, confirm this claim.

The PTC Integrity Modeler, which is presented in subsection 2.2.1, offers the possibility to generate C code automatically from UML/SysML diagrams. It suffers from the disadvantage that the code generator works very slowly even for small models. Additionally, the PTC code generator does not support automatic code synthesis for activity diagrams. For state machines, however, the automatic code synthesis sometimes produces dead code, which is undesirable not only in avionic systems. In addition, it is not compliant with Motor Industry Software Reliability Association (MISRA). The PTC Integrity Modeler’s code generator does not support calls to an operating system or an
abstraction layer such as the OAL.

Sinelabore RT [27] also has developed a code generator for activity diagrams that generates C++ or Java code. They support activity diagrams consisting of initial nodes, final nodes, opaque action nodes, decision and merge nodes, edges and guards [27]. The UML elements will be explained in detail in the following section 2.2. The Sinelabore RT code generator transforms loops, branches, return without parameter, and actions which contain plain C code. With the code generator from Sinelabore RT, important concepts such as hierarchy, message sending and calling functions of a real-time operating system cannot be used.

At the moment, as stated above, no fully-functional code generator for activity diagrams to ANSI-C code with the possibility to generate code for a real-time system exists. This thesis aims at closing this gap.

2.2. Input Format

Definition 1. The Unified Modelling Language (UML) defines a general-purpose graphical notation and it provides several diagram types. The notation elements can be used to model different fields of application. Both static and dynamic issues can be designed [3].

According to Kecher et al., using the UML offers several advantages [3]. Firstly, the notation elements are well-defined and proved by many experts [3]. Additionally, the possibility exists to design every software system nearly completely because the UML is an expressive language. At the same time, the UML is still easy to understand because it represents the system visually. By using different diagram types, the view on the designed system can be varied, and different aspects can be highlighted. Thus, the choice of the diagram type and the focus is set by the programmer. The UML is a language, independent of the target platform and the chosen high-level programming language. Furthermore, the UML is openly accessible and publicly specified by the Object Management Group (OMG).

Definition 2. "The OMG systems Modeling Language (OMG SysML) is a general-purpose graphical modeling language for specifying, analyzing, designing, and verifying complex systems that may include hardware, software, information, personnel, procedures, and facilities." [19]

The set of System Modeling Language (SysML) elements forms an intersection with the elements defined in the UML 2.0. A subset of the UML elements is reused, and some element extensions are introduced. Furthermore, some new diagram types are defined such as the requirement diagram. An overview of the differences regarding diagram types is shown in figure 2.1.

The SysML extends the UML activity diagrams with the possibility to design continuous systems, to support control operators, to accomplish compatibility with Enhanced Functional Flow Block Diagrams (EFFBD), to allow semantics for function decomposition, and to model probabilities [20]. The UML activity diagram elements used in this
thesis are equal to the ones available in SysML. The extensions for activity diagrams made in SysML will not be covered. Thus, the terms UML and SysML can be used synonymously here. In subsection 2.2.2, the required UML elements will be presented individually. Block definition diagrams can be used to visualise the hierarchy of several activity diagrams [6], but they are also not covered in this thesis.

2.2.1. Environment

The PTC Integrity Modeler [22] is a tool provided by the Parametric Technology Corporation (PTC). The PTC Integrity Modeler offers the user an interface to graphically design a system with the UML and the SysML. Additionally, the modeller supports an interface to DOORS, which is developed by IBM to manage requirements [14]. Airbus employees already use DOORS to manage requirements and the integration into the PTC Integrity Modeler allows seamless exchange of requirements. Furthermore, the user can export the model to XMI. XMI will be explained in detail in subsection 2.2.3. As explained in [22], the modeller is also able to generate source code from diagrams automatically. Unfortunately, this is not yet possible for activity diagrams, so the results of this thesis cannot be compared to a commercial implementation. The PTC Integrity Modeler also offers the possibility to embed visual basic scripts as macros to add functionality to projects, packages, diagrams or elements. Also, user profiles can be created to disable particular elements for a specific group of users. This feature can be used to enforce a coding standard among system designers.
2.2. UML Elements

The UML activity diagram is a diagram type that is responsible for modelling the behaviour of a system. An example of an activity diagram can be found in appendix E. Other behaviour diagrams are use case diagrams and state machine diagrams. In a UML activity diagram, the elements of a behaviour get connected via data and control flow edges during the design process [3]. An activity diagram presents "behavior that specifies the transformation of inputs to outputs through a controlled sequence of actions" [6, p.205]. The nodes belonging to an Activity diagram, are called ActivityNodes. Edges between the ActivityNodes are described as ActivityEdges. The order of the ActivityNodes and their dependencies is determined with ActivityEdges. The ActivityNodes can be grouped in the three categories of control nodes, executable nodes and object nodes. Nodes that coordinate the flow belong into the control node category. Executable nodes are ActivityNodes, which perform an action. ActivityNodes that store and manage resources belong to the object nodes [3].

Initial Node  The initial node represents the starting point for the control flow when the activity is invoked. According to the literature [3] [23], it is allowed that one activity diagram can have more than one initial node, while in this implementation only one initial node should be allowed to define a unique and unambiguous starting point. Only outgoing control edges can be connected to the initial nodes. In the case of multiple outgoing control edges, only one flow will be taken. Guards are used to specify which flow to take. In this implementation, only one outgoing edge is allowed. Neither incoming edges nor data flow edges are allowed to be used together with the initial nodes [6]. The graphical representation is shown in figure 2.2.

Final Node  In UML activity diagrams, there exist two types of final nodes which are the activity final node and the flow final node. Their graphical appearance is presented in figure 2.2. The activity final node terminates the flow in the whole activity and thus terminates the activity execution itself. The flow final node just terminates the flow that ends in the flow final node, but parallel flows and thus the activity execution can continue. In contrast to the initial node, control flow as well as data flow can point to a final node but no outgoing edge can exist. It is possible that an activity diagram has multiple final nodes [3] [23] [6].

Decision and Merge Node  Decision and merge nodes are used to model alternative execution paths and loops. Except for their number of incoming and outgoing edges, the graphical representation is in both cases a diamond as shown in figure 2.3. As
described in [3], the decision node branches the control flow. This node type always has one incoming and several outgoing edges. The execution continues on exactly one flow. Guards with disjunctive conditions can be used to make the behaviour deterministic. To avoid that the execution halts, all possible input values should be covered in the set created by the union of the guards. The order in which the conditions are checked is undefined.

Furthermore, [3] explains that the merge node merges the branches which are produced by the decision node to one subsequent flow. Thus, the merge node has multiple incoming edges and one outgoing edge. Only the one flow that was taken at the decision node will be continued in the merged flow.

The UML 2.5 specification [3] allows the combination of a decision and a merge node to a single notation element so that several flows can be merged and the subsequent flow branched at the same time. Therefore, the combined notation element has multiple input and output flows. Instead of using decision and merge nodes, multiple flows can also start or end directly in an action node which is explained below [3]. In contrast to [3], the SysML guide and the PTC Integrity Modeler allow branching and merging data flows as well [6] [23]. However, it is not possible to create a combined decision and merge node with the PTC Modeler in practice.

**Fork and Join Node** Fork and Join nodes are used to model parallel and concurrent execution paths. They are represented in the PTC Integrity Modeler by a black bar as shown in figure 2.3. A fork node has one incoming edge and several outgoing edges. This node type splits the flow analogous to the decision node but instead of creating alternative execution paths, several parallel execution paths exist [3].

A join node synchronises the concurrent execution paths. The designer can specify for each join node, which incoming flows have to reach the node to continue further execution over the single outgoing edge. If nothing is specified, the default join specification is set to ‘and’. That means that every incoming edge of the join node has to be traversed once before continuing.

In [3], the fork and the join node are only used with control flow edges. The SysML guide and the PTC Integrity Modeler also allow forking and joining data flows [6] [23]. As with decision and merge nodes, a successive execution of a fork and a join node can be replaced by a single notation element with multiple incoming and outgoing edges. This is also not possible to design with the PTC Modeler.

![Figure 2.3.: (a) UML decision node, (b) UML merge node, (c) UML fork node (d) UML join node](image)
2.2. Input Format

Action Nodes  An action represents an atomic unit of functionality that can be executed. The actions can consist of multiple statements, of which their consecutive execution would not be atomic as such. But in the model, the grouped statements can be treated as if there is only one action [3]. Actions process input values and produce output values. The input and output values are linked to the action through its pins [6]. Pins are described in detail further below. An overview of the graphical representation of the different action nodes is shown in figure 2.4.

The PTC Integrity Modeler provides the opaque action as a versatile action node. An opaque node is used when the required functionality cannot be expressed by any other UML element. The opaque node contains a description of the action being expressed by source code of the target language. In this thesis, the considered target language will be ANSI-C99.

A specialised group of action nodes are the call operation and the call behaviour action nodes, which are also provided by the PTC Integrity Modeler. According to Friedenthal et al. [6], the call behaviour action invokes any other SysML behaviour. This can be modelled as another activity, a state machine, a sequence diagram, or a use case diagram. The parameters of the invoked behaviour will be represented as pins of the call behaviour action node [6]. The call operation action node is an additional alternative, supported by the PTC Integrity Modeler. This node can be used to 'transmit [...] an operation call request to the target object, where it may cause the invocation of associated behaviour' [23]. An operation represents a function or transformation belonging to a class or interface with a clearly defined signature [23]. With the call action nodes, hierarchies can be applied to structure activities, which leads to a comprehensible design.

Another specialised group of action nodes are the send signal and the accept event action node. These action nodes offer the possibility to model a communication with messages between two activities. The send signal action node asynchronously sends a signal or respectively a message to the corresponding accept event action node, which waits for the signal until it is received. The accept event action starts waiting when an incoming control flow reaches the node. If the node does not have any incoming edge, the node is activated as soon as the owning activity starts to execute. The same signal can be sent by several functions, but only one receiver for that signal may be designed. Signals can be triggered by a send signal action or after a certain specified time has passed [3] [6] [23].

The create object action node is also available in the PTC Integrity Modeler context. An output pin is always generated when placing the create object action into the activity diagram. This node offers the possibility to create an instance of a class in an object-oriented context and provides the object at an output pin. Furthermore, it is possible to generate a variable with an arbitrary supported data type, like int. The output pin then supplies the variable. In addition, a signal that is available in the context of the activity can be created and provided at the output pin.
2. Background

Object Node and Activity Parameter Node

An object such as a variable has to be available at different action nodes for processing. In the UML, there exist two alternatives to model the transfer between two action nodes. Either an object node is inserted or the pin notation is used. With an additional object node between the action nodes, the data flow connects each action node with this newly inserted object node. In the pin notation, each action node gets a pin, and the data flow connects the pins directly. In both notations, the state, in which the object has to be before being transferred to the receiving action node, can be saved [3]. The PTC Integrity Modeler only supports the pin notation to transfer objects. The graphical notation is shown in figure 2.4 (a).

The activity parameter node is a special object node. It represents the interface of the activity to the outside. The node can be defined as an input or an output parameter, and it can be selected whether it is the return parameter. The activity parameter node has a name, a data type and a direction. The parameter node is placed on the edge of the activity like shown in figure 2.5.

Central Buffer and Data Store Node

The central buffer node and the data store node belong to the object nodes and their graphical representation is shown in figure 2.6. They are used to integrate data streams. In PTC Integrity Modeler, a control flow edge can neither be added as an incoming nor as an outgoing edge for a central buffer node or a data store node [23]. Pins are only constructed to connect one producer with one consumer. In this scenario, a producer is an action node with an output pin, and a consumer is an action node with an input pin connected through a data flow edge. In contrast to pins, a central buffer node and a data store node are required if several producers generate a resource or if several consumers want to access a resource. When a consumer needs a resource, the stored resource is removed from the central buffer node while the data store node keeps the resource and just provides a copy to the consumer. Thus, the resource can be used by multiple actions during execution. When a resource reaches a data store node, and the resource is already stored, the value of the resource
will be overwritten. Both data store nodes and central buffer nodes only work while the parent activity is executing [6].

![Diagram of data store and central buffer nodes]

Figure 2.6.: (a) UML data store node, (b) UML central buffer node

**Activity Partitions** The UML has multiple ways to link the activity execution to a context given by a designed block or class. By this connection to other elements, the responsibilities become unambiguous [3]. One possibility is to divide the actions in the activity regarding the context, in which the action should work, by means of activity partitions like shown in figure 2.7. The action nodes are placed in the region with the corresponding partition name, and thus the responsibilities become specific. Alternatively, a block or class can own the activity. The relation between the behaviour and the structure has to be specified. The activity in a block context represents either a method or a behaviour that is described over the lifetime of the block [6].

![Diagram of activity partition node]

Figure 2.7.: UML activity partition node

**Structured Activity Node** "A structured activity node is often used in preference to an activity when its actions are unlikely to be reused in more than one context." [6, p.225]. As described in [6], three types of structured activity nodes exist. The first and simplest one is a sequence structured node, which groups a subset of activity elements together and executes the including actions according to the control flow. Its graphical representation is shown in figure 2.8. The second one is a loop structured node, which repeats the containing actions. It has three sections which are setup, test and body similar to for-loops in the C programming language. Whether the body is executed before the test or after is not specified. The last structured node is the condition node, which contains several subsets of actions and executes them according to a certain condition. Therefore, a set of clauses with each having a test and a body section is required. The else clause is a special clause, which gets executed when all if clauses yield false. Only one body can be executed. In the case that more than one condition yields true, it is undefined, which body is executed, and so the designer needs to specify a rule for the order of execution himself [6]. However, a construct of multiple if-else statements violates the design standard assumed for this thesis and thereby a case with an undefined order cannot occur.
Interruptible Activity Region  The interruptible activity region groups several activity elements like a structured node. The execution of at least one action starts as soon as the owning region is entered. The execution can be interrupted and terminated by means of an exception flow or respectively by an interrupting edge. An interrupting edge is represented by a control edge with an additional flash symbol, and it starts in the interruptible region and ends outside of that region [3]. According to [6], an interrupting edge can also be represented by a data flow edge. An edge that is not an interrupting edge can also leave the interruptible activity region. In contrast to the interruptible edge, these edges do not cause a termination of potentially executing actions inside the region. The interruptible activity region is used when an exception occurs, and multiple nested or parallel processes should be terminated immediately, and/or an error should be handled. Additionally, the possibility to terminate just a subset of activity nodes is offered [3] [6].

Expansion Region  An expansion region, which is shown in figure 2.9, groups several activity elements together like a structured node. The actions inside the region are performed for each value of the passed group of input values repeatedly. The type of each input value in the passed group has to be equal. The type of the input values can differ for different groups. For example, a group of just numbers and a second group of just characters are each valid, while a group of a mix of numbers and characters is invalid. After processing a group of input values, the resulted output values will be available when leaving the expansion region. The output values are in the same order as their corresponding input values. The processing of the input values can occur in parallel, iteratively, or the input values are considered as a single input data stream. The expansion region can be used if each input value of a set has to be processed equally [3]. The PTC Integrity Modeler does not support expansion regions [23].

Activity Edges  Dependencies between the action nodes and their needed data can be modelled by means of the activity edges. Activity edges are directed connections between
2.2. Input Format

activity nodes. The edges can have guards, which determine whether the flow continues on an edge.

The object flow, also named data flow, models the transfer of information or physical items between object nodes. The data can be passed to an action node by means of its owning pin. The connected object nodes must have compatible data types. Additionally, the direction of the object flow edge has to be consistent with the object node’s direction [6]. Furthermore, a weight can be set that defines how many objects can be transferred across that edge.

The control flow determines the order in which the action nodes are executed [3]. "When a control flow connects one action to another, the action at the target end of the control flow cannot start until the source action has completed" [6].

Stereotypes A stereotype specifies the reason or a role of a notation element and can be used within every diagram type. 'Stereotypes add new language concepts, typically to support a specific application domain' [6]. Each activity element can be described more precisely by adding a stereotype. The notation for a stereotype is «stereotype name».

Several stereotypes can be grouped to profiles. The user is able to define new stereotypes to extend the UML [6], [3]. A user-defined stereotype can be restricted to be used only in combination with selected notation elements. Additionally, different attributes can be assigned by means of tag definitions. A notation element can apply to more than one stereotype [6]. With a stereotype, missing information can be assigned.

All the language element listed in this section can be part of a UML diagram that serves as an input to the algorithm developed in this thesis. This means that for each of these elements, a transformation has to be specified as done in appendix A.

2.2.3. XML Metadata Interchange (XMI)

XML Metadata Interchange (XMI) is a standard specified by the OMG [21]. The interchange format can be used to exchange metadata of UML models between different tools. As in XML, the data is organised with tags like `uml:model`, `packagedElement` or `ownedBehavior`. By means of the opening and closing tags, the XMI elements are structured hierarchically like in XML. The UML element properties are encoded in the XMI attributes of the corresponding element tag except for the expression of a guard, which is encoded in the inner text of its body-tag. The important tags for activity diagrams are the following:

- `ownedBehavior` is the tag for each behaviour designed in UML. The attribute `xmi:type` then specifies whether it is an activity.
- `node` is the tag for every UML activity node. To distinguish between the different node types, each node tag also has an attribute `xmi:type`, which determines the specific node.
- `edge` is the tag for both control edge and object edge. As with the `node` tag an attribute `xmi:type` is present, which determines the specific edge.
Each tag within the `uml:model` has an attribute for a unique identifier such that each element can be referenced.

The PTC Integrity Modeler is able to generate an XMI file from a designed UML model automatically [22]. In this thesis, the generated XMI file will be the direct input for the code generator, which is presented in section 3. XMI as input for the code generator offers the possibility to generate ANSI-C code independently from the modelling tool. Unfortunately, the PTC Integrity Modeler 8.2.15 does not export all information, and some information is exported incorrectly.

### 2.3. Output Format

In avionic systems, the requirements for a system are stricter than in other domains. The system has to fulfil the accreditation and safety-critical requirements. Thus, not all C language elements are allowed to be used in avionic programs. The generated source and header files have to comply with the Airbus Software Coding Standards (SCSs) and the MISRA C standard [18]. In chapter 4, the generated C-files are checked against this standards with static code analysis.

In the context of a C program, activities describe the body of a function. To later use the generated files, the function signature needs to be generated, too. In this thesis, the following language elements and structures have to be considered to describe the C function body.

- Local variable declaration and their assignment
- Branches → if...else, switch...case
- Loops → while, do...while, for
- Function calls
- Inter-task communication → message sending and receiving
- Timer
- A non-interruptible region with corresponding locks

For some language elements, the Operation System Abstraction Layer (OAL) functionality has to be considered, for instance in the case of inter-task communication. The OAL has been developed by Airbus employees to be independent of the underlying RTOS. The OAL implements the interfaces to abstract from the concrete implementation. Every function starts with the prefix `OAL_` followed by a name that states the functionality. The OAL provides functions to enable and disable interrupts, to pass messages blocking and non-blocking, to manage timers, to handle semaphores and to initialise the RTOS. Additionally, memory can be managed, and task operations like `create`, `delay`, `suspend` and `resume` can be called.

The Airbus coding standard constrains the C language structures from above. All rules can be found in the MISRA-C coding standard [18] or in the Airbus Software
2.4. Transformation

Coding Standard, which is even more restrictive. The rules that are important for this thesis are listed in table 2.1.

<table>
<thead>
<tr>
<th>No</th>
<th>Rule</th>
<th>MISRA-C Rule number</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>For every if branch also an else branch has to be defined to ensure coverage.</td>
<td>14.10</td>
</tr>
<tr>
<td>2</td>
<td>Keywords like goto and continue are forbidden.</td>
<td>14.4 / 14.5</td>
</tr>
<tr>
<td>3</td>
<td>Multiple assignments like x = y = 2 and multiple return statements are not allowed.</td>
<td>14.7</td>
</tr>
<tr>
<td>4</td>
<td>Each case in switch...case must have a break and exactly one default case must exist.</td>
<td>15.2 / 15.3</td>
</tr>
<tr>
<td>5</td>
<td>It is required to use static allocated memory instead of dynamic allocated memory.</td>
<td>20.4</td>
</tr>
<tr>
<td>6</td>
<td>A variable must be initialised when declared except the extern keyword is used.</td>
<td>9.1 *</td>
</tr>
<tr>
<td>7</td>
<td>A variable declaration has to be at the beginning of a function.</td>
<td>8.6 / 8.7</td>
</tr>
<tr>
<td>8</td>
<td>Function arguments should be sorted beginning by the input arguments and ending by the output arguments.</td>
<td>16.4 *</td>
</tr>
<tr>
<td>9</td>
<td>Function pointers are not allowed.</td>
<td>11 *</td>
</tr>
<tr>
<td>10</td>
<td>A function must have a single point of exit.</td>
<td>14.7</td>
</tr>
<tr>
<td>11</td>
<td>The source code must not contain dead code.</td>
<td>14.1</td>
</tr>
</tbody>
</table>

Table 2.1.: Airbus coding standard rules and corresponding MISRA-C rules; rules marked with a * are stricter than the addressed MISRA-C rule

2.4. Transformation

The code generator needs to implement a transformation, which takes the input and produces code for the target platform [7]. To achieve this, the XMI file needs to be parsed into an internal data format. Then the internal data format has to be processed and transformed into the output format which is stored in a file. With the automatic code generation, the software development process is extended as shown in figure 2.10. In the DO-178, the classical software design process is described. In the beginning, requirements have to be specified. Then a design is created that is compliant with the requirements. Afterwards, this design has to be implemented manually. Finally, the code is integrated into the system. The requirements and the design can be defined model-based instead of based on the textual representation. The guidelines for a model-based process are specified in DO-331. If the automatic code synthesis is qualified according to
DO-330, it can replace the manual implementation. As a prerequisite, the design needs to be specified model-based.

Figure 2.10.: Software Development Process (Detlef Götting)

The challenges of algorithmic automatic code generation are similar to the challenges occurring when implementing traditional a compiler. The compiler code generation step tries to bridge the semantic gap between a high-level language and binary code [24] while in this thesis the semantic gap between the UML and ANSI-C code needs to be bridged. However, compilers typically ignore some information such as the names of symbols (variables, functions). In contrast, the translation from UML sometimes has to generate this information, because it is not present in the model, e.g. the order of function parameters which is not specified in UML.

As an addition to the correct translation, it has to be proven that the algorithm always terminates. Furthermore, the produced code needs to be documented and readable. Additionally, the code has to be traceable and of good quality [1]. In [15], it is stated that "automatic code generation impacts the quality attributes of the software". Generated code is usually unstructured and unreadable for humans and often larger than code, which is implemented manually. However, a good code generator needs to create efficient code [15]. The PTC code generator for state machines produces C code according to specified design patterns, independent of the modelled nodes. This results in dead code. In this thesis, the code generator should only generate code for designed nodes.

The specific challenges in this thesis are to identify branches and loops in the control flow, to trace the data flow, to associate the parameters in a function signature with the ones in the function call and to deal with missing information, e.g. whether an accept event action receives a message blocking or non-blocking or what the default value for a variable is.
3. Code Generation

In this chapter, the code generator is presented, starting with the description of the parser implementation and the internal representation. Then the transformation rules and the developed algorithm are explained. Afterwards, the invariants for the algorithm and challenges of automatic code generation are summarised.

UML diagrams are the first part of the automated process presented here. A designer specifies a program as a UML activity diagram with the PTC Integrity Modeler as already described in 2.2.1. The diagram is then automatically exported to an XMI file by the PTC Integrity Modeler. Afterwards, the parser interprets this XMI file, and the UML elements are transformed into an internal graph representation. Then the algorithm operates on this internal graph and outputs a C source file. This process is shown in figure 3.1.

The code generator consists of a front-end library called *libxmi* written in C#, a C# project for the actual code generation called *ActivityGen*, and a C# project for the unit tests.

3.1. XMI Parser

Two other theses are in parallel to this and cover the code generation from other diagram types and [13], [12]. The front-end library, which is responsible for parsing, is used as a shared feature in all of these works and only an extension to support activity diagrams was added for this thesis. This extension which is responsible for parsing elements specific to activity diagrams is described in this section. The interface of the library provides a class *UmlObject*, the two classes *XmiNodeTypes* and *XmiNodeAttributes* containing dictionaries for data mapping and the two functions *RegisterTranslationAction* and *RegisterPostActions* for the registration of actions when a concrete XMI tag is found in the file.

The XMI structure is hierarchical and object-oriented, so the internal data structure will represent the input with objects. The class *UmlObject* owns the attribute *id* that is mandatory for each tag that belongs to a UML activity diagram within the XMI file. This *id* allows a unique identification of each UML object. In the *libxmi*, a new data class has to be created for each UML activity element. Both activity nodes and edges belong to the UML activity elements. Each class inherits from the class *UmlObject*. An overview of all possible classes is shown in figure 3.2. Every other attribute than the
id that characterise a UML element is then specified in the derived classes. Because of the inheritance, the code generator front-end is easily extensible without modifying any code for parsing.

The classes XmiNodeTypes and XmiNodeAttributes each contain a dictionary. The class XmiNodeTypes maps the values of the attribute xmi:type, e.g. uml:Activity to the corresponding elements of the internal enumeration XmiNodeType. The class XmiNodeAttributes maps the identifiers of the XMI attribute tags like xmi:type, xmi:id, name or visibility to the corresponding internal enumeration XmiNodeAttribute. To enable the parser to recognise a specific UML element in the file, its unique xmi:type and the tag attributes need to be added to the respective dictionaries.

With the RegisterTranslationAction function, actions can be registered that should be executed while reading the XMI file. The registered translation actions are executed when the associated XmiNodeType matches a tag in the XMI file. These actions then create new objects which represent UML language elements and inherit from the UmlObject class. A list of UmlObjects stores the objects that are constructed while reading the document. The RegisterPostActions function registers actions that are performed after completely scanning the XMI file. While iterating the list of UmlObjects, the registered post-actions trigger for all objects that match the action’s associated type. These post-actions are required to resolve dependencies between two objects like a UML pin that is mapped to a UML action node. The order of the elements in the XMI file creates the need for a separate post-processing step that executes the post-actions because some elements depend on information occurring later in the file. One example of such a dependency is the information about pins which is stored in a separate tag succeeding the corresponding action node in the file, and thus this information is not available during the construction of the action node. During the post-processing step all XMI elements have been read and registered once, so the creation of links between different entities is possible. The actions’ implementation are part of the ActivityGen C# project. The process of using and converting the now stored information is explained in this chapter in the sections below.

<table>
<thead>
<tr>
<th>UMLActivity</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ InitNode: UmlInitialNode</td>
</tr>
<tr>
<td>+ Edges: List&lt;UmlFlow&gt;</td>
</tr>
<tr>
<td>+ Nodes: List&lt;UmlObject&gt;</td>
</tr>
<tr>
<td>+ Parameter: List&lt;UmlActivityParameterNode&gt;</td>
</tr>
<tr>
<td>- IsReentrant: bool</td>
</tr>
<tr>
<td>+ UMLActivity(xmi: XmiNode)</td>
</tr>
</tbody>
</table>

Figure 3.3.: UML activity class

An UMLActivity class (3.3) owns three lists, one for the nodes, one for the edges, and one for the parameters belonging to the activity. Additionally, one variable stores the initial node, which marks the entry point of the UMLActivity class. While the translation actions are executed, object and control flow edges are added to the list.
of edges and activity parameter nodes are added to the parameter list. Furthermore, action nodes, data store nodes, and control nodes except for the initial node are added to the list of nodes. The initial node is a special case because it should appear only once in an activity. To ensure this, an activity owns a variable that stores the initial node separately. The reason is that the programming language C belongs to the procedural languages, which have a sequential execution with a single starting point. According to the coding standard, only one return statement may exist in a C function. Because a final node cannot carry information about the return parameter, the return statement is outsourced into an opaque node. Thus, the number of final nodes does not tell anything about the number of return statements and therefore a final node will be added to the list of nodes. All other UML elements will be assigned to the elements in the activity lists while the post-actions are executed. An overview of all possible UML element types is shown in figure 3.2. The flag IsReentrant signifies whether an activity may be invoked recursively or by a parallel process while one instance is still in execution. Essentially, it states if an activity has resource constrains that would make reentrancy undesirable or impossible. Recursive invocation is excluded because the modelling standard forbids recursive calls to actions and in the future, a model checker will prevent the design of models using recursion. Enforcing that a block is not entered while it is still running requires the use of locks. Even without recursion, this can lead to deadlocks blocking the execution of all paths. Since the PTC Integrity Modeler only allows reentrant activities, the solution to this problem is regarded outside the scope of this thesis.

While parsing the XMI file and transforming it into the internal representation, it is ensured that the right number of edges is assigned to each node. Every action node and every structured node must have one incoming and one outgoing control edge. Only control nodes are allowed to have more control edges, e.g. a decision or a merge node. The initial node may only have exactly one outgoing and no incoming control edge. The final node may respectively only have exactly one incoming and no outgoing control edge. Both initial and final node are not connected to data flow edges. Control flow edges cannot be used with data store nodes. The PTC Integrity Modeler restricts the number of data flow edges at pins to at most one, and while parsing it, an error message is logged if no data flow is connected to a pin. Data store nodes are not limited in the number of data flow edges. Action nodes and structured nodes cannot be connected with a data flow edge directly. For control nodes in combination with data flow edges, the same restrictions as for control flow edges apply.

### 3.2. Internal Structure

After interpreting the XMI document, the collected data needs to be represented by an internal structure that connects the single elements within the UMLActivity class. The collected data originates in a UML activity diagram, and since the elements are represented as a graph in the UML diagram, the internal structure will also be a graph. With this graph structure algorithms from graph theory can be used to find branches and loops and to distinguish between them. The open source library QuickGraph [4]
provides classes and algorithms to handle graphs. The library offers interfaces for mutable and immutable graphs. Since the algorithm operates on the graph by changing nodes and edges, an immutable graph is unsuitable. In QuickGraph four types of mutable graphs exist. The AdjacencyGraph only offers an interface to access the outgoing edges. The BidirectionalGraph can also access the incoming edges. For sparse graphs, the BidirectionalMatrixGraph can be used, and for undirected edges, the UndirectedGraph is offered. The algorithm requires a directed graph with access to incoming and outgoing edges and activities are often abundant graphs. Hence, the BidirectionalGraph is used.

An activity contains both control flow and object flow edges. To reduce the complexity, a BidirectionalGraph with control flow edges and another BidirectionalGraph with object flow edges but the same nodes are used. The Graph class, shown in figure 3.4, encapsulates these two BidirectionalGraphs and offers methods for operating on the graph. Both BidirectionalGraphs are initialised as empty graphs at the beginning of the transformation.

```csharp
+ ActGraph: BidirectionalGraph<UmlObject, UmlEdge>
+ ObjGraph: BidirectionalGraph<UmlObject, UmlEdge>
+ Graph()
+ Replace(oldNode: UmlObject, newNode: UmlObject)
+ combineCNodes(adjacentSourceNode: CNode, adjacentTargetNode: CNode): UmlObject
+ Act2Graph(Nodes: List<UmlObject>, Edges: List<UmlFlow>, InitNode: UmlInitialNode)
+ DepthFirstSearch()
+ printGraph(path: string)
+ ToGraphML(): string
+ printVerticesAndEdges(): string
```

Figure 3.4.: Graph class

To insert the activity nodes and edges into the graph, the function Act2Graph is offered by the Graph class. The Act2Graph function is called for each activity individually. Thus, for each activity, an own graph exists in the internal representation. The QuickGraph library provides interfaces to easily insert the activity nodes and edges to the BidirectionalGraphs within the Graph class. These BidirectionalGraphs are successively passed as input for the algorithm, which will be explained below.

For the Algorithm 1 an additional node type needs to be introduced. This node type is called CNode and also inherits from the UmlObject class as the UML elements do. A CNode encapsulates only a string that contains the C code for the output file. The algorithm will replace every activity node or a group of nodes except for the initial and the final node by a new CNode. The string within the CNode contains the generated C code. The C code is generated according to the transformation rules explained in appendix A. The initial and the final node result in no C code statement. Thus, no CNode needs to be generated for these two nodes. It would be possible just to leave these nodes out in the internal representation, but with them, a defined start and end point are available in the internal graph. The root node for the depth-first search used later will unambiguously be the initial node. When writing into the output file, however,
3.3. Algorithm

only the single remaining CNode will be considered.

The Replace function takes two nodes from type UmlObject and replaces one node by the other. The replacing node is the new node and thus must not be part of the graph before entering the Replace function but after exiting it. The node that should be replaced is the old node and needs to be in the graph before and should be removed after the replacement. The source of the outgoing edges and the target of the incoming edges of the old node will be set to the new node. The Replace function is an important utility function that will be used during all steps of the transformation, to replace high-level code constructs like an action node with their corresponding C implementation in a CNode. The Replace function requires exactly two nodes, but for generating a branch or a loop a group of nodes has to be replaced by one new node. Hence, the function for the generation of branches and loops will not use the Replace function but a specialised implementation where the incoming and outgoing edges are extracted from the group nodes. For the combination of two CNode, the Replace function does not fit either because two nodes had to be replaced by one. Thereby, it has to be considered that besides the relocation of the edges and the exchange of the nodes in the internal graph representation also the content of the two CNodes has to be concatenated, so the CombineCNodes function can be used.

The replacement of a node or a group of nodes only operates on the BidirectionalGraph with the control flow edges. Due to the incomplete modelling of data flow edges with the PTC Integrity Modeler, the BidirectionalGraph with the data flow edges will only be used for checking and thus it is not changed during the execution of the algorithm.

3.3. Algorithm

As the first part of the algorithm’s description, the mapping between the input UML elements and the output ANSI-C elements is presented by the transformation rules given in appendix A. Only elements explicitly contained in the model are translated to C code through these rules. The code generator does not produce C code for not existing functionality such as the PTC code generator does for state machines. Thus, the code generator does not produce dead code which is prohibited by rule 11 in table 2.1. Nevertheless, it cannot be enforced, that a user refrains from modelling guards that always yield true which would result in unreachable else branches. Also, placing dead C code in opaque actions cannot be prevented. In the future, a model checker will be implemented which will check the C code within opaque action nodes separately against MISRA before starting the code generator.

A hierarchy structure, where the activity diagram is defined within a class, block or operation diagram is used to define in which context the functionality is executed. So the activity partition node will not be considered. In object-oriented target languages, this node could have been used to specify, which instance of an object executes the functionality provided by an activity diagram.

The fork and join node are responsible for the creation or termination of a task. Since the SCS does not allow the dynamic creation of new tasks which might require dynamic
Algorithm 1 Code Generation

Require: graph contains exactly one initial node
Require: graph contains exactly one activity final node
Ensure: graph contains exactly three nodes

1: function COMPUTE
2: createCLocalVariables()
3: transformStructuredActivityNode2C()
4: transformAction2C()
5: graph.DepthFirstSearch()
6: repeat
7: oldNodeCount ← graph.Vertices.Count()
8: mergeCNodes()
9: mergeBranches()
10: mergeLoops()
11: until oldNodeCount ≤ graph.Vertices.Count()
12: return graph

memory allocation, these nodes will not be considered in the algorithm, too. Nevertheless, a concept for transforming these nodes has been developed. A fork and a join node only appear together. An abstract structure of a model containing a fork and a join node is shown in 3.5.

Figure 3.5.: A fork expands the main task by the parallel tasks Task1 and Task2. The join node joins these tasks back to one. The blue edges show the flow the current task takes.

To implement the fork node, new tasks need to be created dynamically. The OAL provides a function TaskSpawn(uint32_t taskNameId, uint8_t priority, uint32_t stacksize, void *p_arg, void(*task)(void *p_arg)) for that. Two challenges arise when planning the concept for a fork node.

The first challenge occurs by using the provided OAL function, which needs a function pointer as one argument. The two options are to create only call action nodes on the parallel paths or to automatically generate a function for each parallel path with its
actions within the code generator. The disadvantage of automatically generating a function is that it becomes difficult to determine which variables need to be passed to the new function. The safe but expensive way is to make every declared variable available. The other option is to collect just the variables that are used in the action nodes on the path. Therefore, it can be iterated over the input pins of the action nodes. The problem is that variables that are used on guards will not be covered reliably. In case that only one call action node on each path is used it is clear which parameters need to be passed. But it has to be ensured that a call action node exists on each parallel path.

The second challenge is to identify on which path the actions are executed in the current task and which paths result in the creation of a new task. A safe and deterministic solution is to create a new stereotype that can be applied to the outgoing edge of the fork node that is on the execution path of the current task. Another approach is to choose an arbitrary path as the main path, which would not be deterministic for the designer. The third approach is to select the path as the main task, whose source of the outgoing edge of the fork node is not a call action node, but that assumes that the designer does not accidentally create two or zero paths without a call action node.

In the concept for implementing a join node, the main problem is to synchronise multiple executions. The OAL offers functions to create and handle semaphores. This offers the opportunity to solve synchronisation with semaphores. However, each parallel path would require its own semaphore. The main task would then be able to continue its execution after the join when it had acquired all of these semaphores. To reduce the number of semaphores one counting semaphore could be created. When all tasks release their virtual resource, the execution can continue. The problem with semaphores is that they have to be defined globally if pointers should be avoided. With activity diagrams, it is not possible to define global variables. If only one parallel path exists, signal sending and receiving can be used to synchronise both tasks. Another possibility is to specialise on one operating system like \( \mu \)C/OS, which provides event flags. Event flags can be implemented by a variable that consists of multiple bits which can be accessed independently. For each parallel task, one bit is reserved. After finishing a task, the flag can be set to one. Then the main task can continue the execution when the specified bit pattern occurs. Another solution could be to change the default join specification to or, so only one task needs to finish its execution to continue, but that would limit the semantics that a designer may express.

In the following, the actual implementation will be described. Therefore, the compute function (Algorithm 1) will be explained in detail. The algorithm generates source code which corresponds to the body of a function. The function’s signature is added when the algorithm has terminated. The compute function will operate on the internal graph structure explained in section 3.2.

At first, the single nodes that are not dependent of other nodes each will be transformed into a corresponding CNode. Such nodes are the data store nodes, structured nodes, and action nodes. Afterwards, complex groups of nodes will be transformed into a single CNode within a loop until the algorithm terminates. The algorithm will terminate successfully if and only if at most three nodes are left, or it will terminate faulty if this
condition is not met and the total number of nodes has not been reduced during the last iteration. The nodes left should be one initial node, one final node and at most one CNode containing the C code for the considered activity. When the algorithm terminates, the internal graph representation will be returned, and the constructCFileContent function makes sure that the code of the CNode is written into a file and that the activity parameter nodes will be considered to generate a signature. In section 3.4, this function to write into the output file will be discussed in detail.

2: createCLocalVariables()

The algorithm starts by generating the code for declaring local variables. In UML a local variable can be modelled with a data store node. Alternatively, a create object action node can be used. Because both nodes cannot express enough information to define a default value, a constant, a pointer, an array or a bitfield, a stereotype is needed to extend the node. With the existing CLocalVariable stereotype a default value can be modelled, which is required in table 2.1 rule 6. In addition, pointer variables, constants, arrays, local anonymous structs with bitfields and C storage classes can be designed. Therefore, the designer can either select a value from a given set such as the C storage classes or insert a value into a text box. In this thesis, it is assumed that the designer enters only valid C code. To be able to model all required information this stereotype is mandatory. The create object action node is mainly used for object-oriented target languages and it is a special action node in the PTC Integrity Modeler. In the past, data store nodes in combination with the CLocalVariable stereotype are used to model local variables. Thus, this combination will be defined to used in a model to specify an ordinary local variable. Since no control flow edge can be used together with a data store node, the data store nodes are modelled free-standing in the activity diagram. If a variable is used to set the default value for another variable, the dependencies can be represented by means of data flow edges. The end of the edge points to the variable that uses the other variable. In the algorithm, the data store nodes get sorted in topological order, so the dependencies between the data store nodes are resolved. A topological sort operates on a forest (set of trees). At first, nodes without any incoming edges are taken, because they do not depend on any other node. These nodes are appended to the list of sorted nodes, and they and their outgoing edges are removed from the forest. Thus, a new set of nodes will have no incoming edges, and the topological sorting can continue until the forest is empty. A general description of the topological sort can be found in [28]. As a precondition, no cyclic dependencies may exist between the data store nodes. Otherwise, an exception is thrown because the assumption of a forest is not given. In the future, a model checker will be developed, which will be responsible for checking this precondition. At the moment a validation of this precondition results in an exception as shown by the unit tests in section C. The createCLocalVariables function takes all data store nodes from the internal graph in topological order and replaces them by a single CNode. An example graph is given in 3.6.
3.3. Algorithm

The graph in figure 3.6(a) results in a string, where first variable $i = 0$, then $j = i$ and finally $k = j$ is declared. For the graph in (b), an exception is thrown, because the variable declarations cannot be transformed into a valid string, due to cyclic dependencies. The resulting string is assigned to the string encapsulated by the CNode. Then this CNode is added to the graph. Since the stereotype offers different fields to define a variable, the following expressions, which use regular expression notation, show how the information from the stereotype is used to form C statements that can be assigned to the string encapsulated by the CNode.

\[
\text{(static\|auto\|register)?\ (const\|volatile\|const\ volatile)? \ type} \\
\quad \text{(indication)? \ name ( \ [1 - 9] + \ )? = default value} \tag{3.1}
\]

\[
\text{(extern)?\ (const\|volatile\|const\ volatile)? \ type (indication)? \ name ( \ [1 - 9] + \ )?} \\
\quad \text{(static\|auto\|register)?\ (const\|volatile\|const\ volatile)? \ struct \{} \\
\quad \quad \text{(bitfields) + } \ name = default value \tag{3.2}
\]

\[
\text{(extern)?\ (const\|volatile\|const\ volatile)? \ struct \{ \ (bitfields) + \ } \ name} \tag{3.3}
\]

The user selects the storage class value and the const/volatile qualifier from a given set of values in a drop down menu. The type is selected from a predefined set of values in a separate window. The other fields are all text boxes in which a user can enter an arbitrary text. A pointer variable can be modelled by inserting an asterisk character (‘*’) into the text box titled indication. For this thesis, it is assumed that the text contains valid C code, but in the future, a model checker has to prove that the text boxes only contain valid code.

Function pointers can also be set in the stereotype. Because function pointers are forbidden by the software coding standard (table 2.1, rule 9), this input needs to be caught. As an additional restriction posed by the coding standard, a default value has to be set except for when the C storage class keyword extern is used, which is specified in table 2.1, rule 6. This constraint is passed to the designer who has to provide an initial value for all variables used in the model. Otherwise, the code generator throws
3. Code Generation

an exception. Also, default values must not contain an equal sign to avoid multiple
assignments as described in table 2.1, rule 3. In case an array is specified, the default
value needs to be enclosed in curly brackets. Enumerations and structs should be defined
in the enclosing class or block diagram to be known globally. Activity diagrams can only
introduce variables that are local to a single function. The MISRA standard demands
that all variable declarations are placed at the beginning of a function body before any
other expression (see table 2.1, rule 7). This is solved by inserting the generated CNode
between the initial node and its successor. In case no data store node is modelled, the
algorithm inserts a CNode with an empty string.

3: transformStructuredActivityNode2C()

In the next step, structured activity nodes will be transformed into a new CNode. Within the transformStructuredActivityNode2C function, structured nodes are divided
into three cases: a sequence structured node, a C for-loop and a structure node with
a newly developed stereotype that provides a non-interruptible region. Each structured
node can contain a subgraph with an own initial and final node and the functionality
that should be expressed with this structured node. At this point, the subgraph is not
part of the main internal graph representation because it is hidden by the structured
node. An own internal graph represents each subgraph. As for the activity, the Graph
class function Act2Graph can be used to transfer the UML elements within the structured
node into an own internal graph. For the internal graph representing the subgraph, the
compute function is called recursively. Due to the specification, each variable needs to be
defined at the beginning of a function, so no data store nodes are allowed to be modelled
within any structure node. If the algorithm terminates successfully, exactly one CNode
remains, since a structured node without expressing functionality does not need to be
modelled and more than one CNode would have been merged. In a sequence structured
node this CNode just replaces the structured node in the internal graph representing the
activity.

In structured nodes which are tagged with a stereotype for an C for-loop, the string
within the CNode needs to be adjusted. Therefore, the designer has to specify the for-
loop initialisation, the condition, and the increment offered by the stereotype. This
input data will be used to build the string for the C for-loop, which looks like for(loop
initialisation; condition; increment). This string will be concatenated with the string of
an opening curly bracket, the CNode, and a closing curly bracket. Finally, the adjusted
CNode replaces the structured node in the internal graph representing the activity.

The structured node tagged with the non-interruptible region is used to design func-
tionality that must not be interrupted by the underlying operating system. Therefore, in-
terrupts are disabled when entering the structured node tagged with the non-interruptible
region and enabled when leaving the structured node. Every action that is designed
within this structured node will be performed without being interrupted. Since inter-
rupts can be enabled and disabled by the operating system, the OAL is used to model
the behaviour. For the non-interruptible region, a new stereotype was developed in this
thesis. The stereotype provides an input field to set a variable, in which the settings of
the interrupt register will be stored temporarily. The OAL function call for disabling interrupts is inserted at the beginning of the CNode string and the OAL function call for enabling interrupts is added at the end of the CNode string. After the adjustment, the changed CNode replaces the structured node in the internal graph representing the activity. This structured node is not described in the literature but is developed in this thesis to offer enabling and disabling interrupts.

The interruptible region, presented in the literature by Kecher [3] or Friedenthal et al. [6], is used to interrupt the action for instance in the case of an error. In object-oriented programming languages, this region is used to model exceptions. Since exceptions cannot be formulated in the C programming language due to rule 2 table 2.1, this model element is not considered in the algorithm. A user profile for designers of a model should be created that disables choosing the interruptible region in the PTC Integrity Modeler.

Another structured node presented by Kecher [3] or Friedenthal et al. [6] is the expansion region. This region is by the PTC Integrity Modeler not provided by default, so the region cannot be transformed by the code generator. The expansion region could be rebuilt with a stereotype, but it is not necessary as a stereotype already exists for a C for-loop. The expansion region would have been transformed into a C for-loop, which operates on an array. The alternative for-loop stereotype is more intuitive and generic since it can model operations on arrays but also on other data structures that exist in C.

It is also possible to add multiple stereotypes to a structured node. Since only a C for-loop stereotype and a non-interruptible region stereotype are present in this thesis, only applying these two has to be considered. In the case both stereotypes are applied, C code is generated that contains a for-loop within a non-interruptible region. If the designer intends to disable interrupts within the body of a for-loop, a second structured node tagged with the non-interruptible region has to be constructed within the structured node with the C for-loop stereotype applied.

4: transformAction2C()

Afterwards, each action node is transformed into a CNode by calling the transformAction2C function. To distinguish the different types of action nodes and to invoke their individual transformation, the visitor pattern is applied. This pattern is commonly used to extend a class hierarchy with polymorphic functions, without the need to modify the class hierarchy itself. Further information on the visitor pattern and an implementation suggestion can be found in the book written by Gamma et al. [8]. With this implementation, each action node has its own individual transformation function outside of the libxmi.

The transformation for an opaque action node assumes that the body of the opaque action node contains only valid C source code that is compliant with the MISRA standard. Then, the body is just inserted into the CNode string attribute. Finally, the CNode replaces the opaque action node in the internal graph structure. The plan for the future is to validate the C code within the opaque action node with a model checker.

Both the call behaviour and the call operation action node results in a function call
in \( C \) after their transformation. According to the UML specification, a \textit{call behaviour node} is able to call every behaviour diagram, but this thesis only focuses on calling other activities. In this thesis, each activity is transformed into a separate \( C \) function with the name of the activity as the function name. The function will be defined when the called activity is transformed into source code.

To deterministically assign the order of parameters in a function's signature, and to fulfill the coding standards, the parameters are always sorted by their direction and alphabetically by their name. The output pin that returns the return value of an activity or operation has the same name as the called function. It is not possible to change the name of an output pin containing the return value in the PTC Integrity Modeler. In \( C-99 \), it is possible to assign the same name to a function and a variable. Nevertheless, it is bad coding style and non-compliant with the MISRA coding standard. Another possibility is to have no explicit return value but an ordinary output pin which implicitly keeps the return value in the parameter list. A second approach could be to add a prefix to the return value name within the code generator. Finally, the \textit{CNode} contains a \( C \) function call with a sorted parameter list independently whether a \textit{call behaviour node} or a \textit{call operation node} is replaced. Alternatively, the implementation can transform just the \textit{call operation node} into a \( C \) function call, and the \textit{call behaviour action node} inlines the generated code for the called activity. The advantage is to have fewer function calls, which can improve the performance. Additionally, it can lead to a more comprehensible representation, if the possibility exists to split up an activity diagram into multiple ones. On the other hand, functions in the source code can become long and redundant. In the case of implementing the \textit{call behaviour node} as a function call, separate activity diagrams in the design result in separate functions in the source code, which can improve comprehensibility. In any case, the \( C \) compiler itself can make the decision whether or not to inline a function in a later step of the translation process.

The \textit{send signal action} and the \textit{accept event action} belong to the type of \textit{action nodes}. Thus, both nodes need to have exactly one incoming and one outgoing edge. The \textit{send signal action} starts sending, and the \textit{accept event action} starts receiving a signal when the node is reached by the control flow. The \textit{send signal} and \textit{accept event action nodes} are used to pass messages between different tasks. To implement the message passing in \( C \), the OAL provides different function calls into the underlying RTOS. The OAL supports both blocking and non-blocking reception of messages but in the UML only one notation element, the \textit{accept event action node} exists. In addition, the UML elements cannot specify the function parameters for the OAL function call. Thus, new stereotypes need to be created in this thesis that serve as an interface to set the OAL function parameters, and that can distinguish between a blocking and a non-blocking reception with a stereotype attribute.

A timer is modelled with an \textit{accept event action node}. In the PTC Integrity Modeler, the user can specify the time when a time event should occur. According to the definition presented in the background chapter, a node is activated when the control flow reaches the node. So the timer starts counting when reaching the \textit{accept event action node}. The outgoing control flow edge is taken when the timer runs out. The timer is set by a function call to the RTOS with an OAL function. The function parameters that are
3.3. Algorithm

required can be assigned by a stereotype.

The create object action node has already been addressed in the paragraph about the creation of local variables. This node is an action node and requires exactly one incoming and one outgoing edge. The create object action creates a local variable at the position in the source code where the action node is placed in the activity diagram. Thus, it is advised to use a data store node instead of this action node to create a local variable as discussed in the paragraph about the creation of local variables.

At this point, each node that does not depend on other nodes has been replaced by a CNode. The resulting CNodes must have exactly one incoming and one outgoing edge. In several iterations, CNodes are combined, and UML nodes are transformed to CNodes until the number of nodes in the predecessor iteration does not differ from the number of nodes in the current iteration.

8: mergeCNodes()

At first, the mergeCNodes tries to combine two adjacent CNodes in the internal graph. To find adjacent nodes, a list of all CNodes in the graph is created and processed. In each iteration always the first node in the list is considered until the list is empty. In the beginning, it is checked whether the successor of the first node is also in the list so they can be merged immediately. If this is the case, the strings of both CNodes are concatenated and assigned to the string of a new CNode. The string from the successor node is added to the end of the first node’s string. If the successor of the first node is not in the list, but its predecessor is, their strings are also concatenated to a new CNode whereby the first node’s string is added to the end of the predecessor’s string. The new CNode is immediately added to the list of CNodes and it is inserted into the internal graph, replacing the first node and its successor/predecessor which will be removed from both. Then the next CNode can be considered. In case that neither successor nor predecessor are in the list, the CNode is removed from the list.

5: graph.DepthFirstSearch()

Branches and loops in the internal graph correspond to loops in an undirected version of the same graph. To find these loops, a depth-first search is performed on the graph. During the execution of the depth-first search, each edge is classified. An edge can be either a tree edge, a forward-cross edge or a back edge. A detailed description of the classification process and of the different edge types is given by Turau [28, p. 98]. A forward-cross edge identifies the execution of alternative paths and the back edges indicate loops within the directed graph, so they can be used to construct branches and loops in the source code, see figure 3.7. When merging multiple nodes into one CNode, the edges keep their classification, so the depth-first search has to be executed just once for every activity.
3. Code Generation

Figure 3.7.: (a) A branch which will be transformed into an *if-then-else*, (b) a branch which will be transformed into a *switch-case* (c) a loop which will be transformed into a *while-loop*. The black edges are tree edges, the green edges are forward-cross edges, and the red edges are back edges.

9: mergeBranches()

In function `mergeBranches`, the algorithm checks if a group of nodes can be combined to generate a C branch. The forward-cross edge just indicates a branch, but in C this can be an *if-then-else* or a *switch-case*. The number of outgoing edges of a `decision node` helps to distinguish between them. A `decision node` with exactly two outgoing edges results in an *if-then-else*. If more than two outgoing edges exist, the elements will be translated into a *switch-case*. In case only one outgoing edge exist, the depth-first search will not find a forward-cross edge and the algorithm will not terminate successfully. This is in line with the coding standard, due to the fact that a *switch-case* always requires a default path in addition to a case and an *if-then-else* always requires an *else*, so the dangling else problem will not occur. Additionally, the number of outgoing edges of the `decision node` has to be equal to the number of incoming edges of the corresponding `merge node`. Only a subgraph with one `decision node`, one `merge node` and at most one `CNode` on each path like shown in 3.7 (a) will be translated. So the innermost structures are combined first. For instance, if a loop is designed in one path of the branch, more than one node lies between `decision` and `merge node`. Thus, the structure will not be transformed in the current iteration.

For an *if-then-else* a condition on one edge and statements on both paths are required. The string within the new `CNode` looks like

```c
if ( condition ) { ifNodeStr } else { elseNodeStr }
```

according to rule 1 in table 2.1. On one outgoing edge of the `decision node`, a guard need to be placed. It is assumed that the guard is a valid C code condition. The guard on the other outgoing edge needs to be either 'else' or empty. On the path, which follows the edge with the condition, a `CNode` is required. The `CNode` contains the statements
for the if block in the string attribute. If a CNode also exists on the other path, its string contains the statements for the else block. According to the coding standard, an else block is also required in case that no CNode is on the else path. For that case, the elseNodeStr contains a 'NOOP;'.

For a switch-case a switch variable, at least two cases and a default case are required as specified in table 2.1 rule 4. If only a single case and the default case are specified, the branch should be redesigned into an if-then-else. The code for the new CNode has a structure such as

```latex
switch (expression) {
    case guard: CNodeStr break;
    ...
    default: CNodeStr break;
}
```

To identify the switch variable a stereotype needs to be applied to the decision node. It is assumed that the designer only inserts a C compliant expression into the text field for the switch variable. For a switch-case exactly one outgoing edge of the decision node has to be indicated as default edge. A guard on a default edge is either 'default' or empty. To select a path as the default path, a stereotype is needed to specify which edge is meant to be the default path. In this thesis, a stereotype called SwitchDefaultFlow is developed to provide this functionality. The stereotype is applied to the control flow edge that the designer chooses as the default path. If not exactly one default edge is designed, an IllegalDefaultPathNumber exception is thrown. A guard for a path which is not the default path is mandatory. The guard contains the value used for the switching decision. If one CNode is on the path, its content is inserted in place of the CNodeStr. Otherwise, the CNodeStr is left empty. If the path is not the default path and the case is left empty, a warning is logged that an empty case is detected and that this case does not fulfil the robustness criterion. If the designer leaves the case intentionally empty, an opaque node containing a "NOOP" has to be inserted while modelling. The cases are assumed not to be overlapping, and each case has to end with a break. Thus, the case order is irrelevant.

Each newly created node for an if-then-else or a switch-case replaces a group of one decision node, one merge node, and the CNodes on each path between those two control nodes. By replacing this group of nodes with one CNode, it becomes possible to combine surrounding structures and adjacent CNodes in the next iteration or in future steps of the current iteration.

10: mergeLoops()

The last step in each iteration checks whether a group of nodes can be transformed into a loop. In C exist three types of loops: for-loops, while loops and do-while loops. The for-loop can be modelled with the structured nodes and the stereotype described above. While and do-while loops can be designed with one decision node, one merge...
node and at most two CNodes like shown in 3.7 (c). A loop differs from a branch by the fact that back edges are produced during the depth-first search. As with branches, the innermost loop structures are transformed first. A decision node has exactly two outgoing edges. One edge belongs to the loop and the other shows where to exit the loop. The loop edge needs to have a guard defined to decide when to stay in the loop. To identify the nodes that are part of the loop, the loop is traversed backwards starting with the source node of the back edge. Every node belongs to the loop that is visited until the target of the back edge is reached. It is the innermost loop if at most four nodes form the loop. There exist four cases to consider whether to use a while or a do-while loop. The first case is if a CNode is placed on both connections between merge and decision node. Then a while loop will be generated. If only a CNode exists on the link from the decision node to the merge node a while loop is created, too. If only a CNode is placed on the connection from the merge to the decision node, it is reasonable to generate a do-while loop. The last case occurs if no CNode exists between merge and decision node. Then an exception (NoLoopBodyException) is thrown. The loop could be an endless loop, since no assignment and therefore no increment within the condition is allowed due to rule 13.1 in [18].

A while loop has the following structure in a new CNode.

\[
\text{doBody while ( condition ) \{ whileBody doBody \}}
\]

A do-while loop is structured like

\[
\text{do \{ doBody \} while ( condition )}.
\]

The newly created node replaces the single decision node, the single merge node and the CNodes. Thus, it becomes possible to combine other nodes as with the translated branch.

3.4. Back-End

In a code generator covering all activity diagram types, an activity would only result in a function body. Then this body has to be inserted at the right position in the right file according to the operation, class or block the activity belongs to. In this thesis, every activity results in a function consisting of a signature and a body. The function name is derived from the activity name. Each function is written into the same C file one below the other. While parsing the XMI file, the activity parameter nodes are added to the activity parameter list. These parameters are used to build the function signature. It is required that a data type is assigned to every activity parameter node. A function always has to define the type of the return value. Therefore, an activity parameter node can be marked as the return parameter and its type becomes the return type. If no activity parameter node is specified as return value, then the return type becomes void. If more than one return value is designed, an exception is thrown. To automatically match the parameters in the signature with the parameters that are set when calling another
activity, the order of the parameters has to be the same. To achieve this, the activity parameters are sorted by their direction and alphabetically by their name as already mentioned in the paragraph about transforming action nodes into C (code line 3.3). The body of the function has been already translated in the execution of the algorithm. Since the algorithm ensures that at most one CNode remains after the algorithm has been run, all code for the function body is stored in the string attribute and can be simply inserted in the structure looking like

```
returnType activityName ( parameterList ) { body }.
```

Finally, the System.IO.StreamWriter is used to write the generated C code into a file, and the formatting is done with GNUIndent [5].

### 3.5. Example: Greatest Common Divisor

To visualise the algorithm, an example is presented in this section. The graph in figure B.1 shows the model of the GCD implementation in the internal graph representation. It is modelled after algorithm 2.

The first step of the algorithm replaces all data store nodes by a new CNode with the variable declaration shown in figure B.2. A CNode is represented by a node with a blue border. Since the variables do not depend on each other, no data flow had been modelled between the data store nodes. Any order of the variable initialisation is a valid result of the topological order, and thus an arbitrary order has been selected.

The next step of the algorithm replaces all structured nodes. Since none structured nodes exist in this example, the internal graph does not change.

Then all action nodes are replaced by CNodes. For all opaque action nodes used, their contained string is copied into the string of the new CNode as presented in figure B.3.

Next, the edges are coloured with the depth-first search, shown in figure B.4, to identify branches and loops in the next steps. Forward-cross edges are coloured green, back edges are red, and tree edges stay black.

In the first iteration of the loop, no CNodes are adjacent, so the graph after mergeC-Nodes stays the same.

Then the forward-cross edges (green edges) are identified to find the branches. In this example, two branches exist, but only the innermost branch fulfils the requirement of a structure with a decision, a merge node and at least two CNodes as shown in figure 3.7 (a). After combining these nodes to one new CNode containing an if-then-else, the graph in B.5 is created. When combining the nodes, the edge classification is not changed. The other branch still does not fulfil the required structure, so no further nodes are merged.

Then the loops which fulfil the requirements as shown in figure 3.7 (c) are transformed. The back edge (red edge) indicates the loop. Because the inner most branch had already been combined, the nodes can be merged to a new CNode containing the if-then-else within the new created while loop as shown in figure B.6. No further back edges exists that need to be transformed.
Because the node count in the first iteration is less than the node count at the beginning, a second iteration will be executed. In this iteration, two adjacent \textit{CNodes} exist, which will be combined to the graph in figure B.7.

Now the remaining branch can be transformed, so the graph in figure B.8 results.

Since the node count between iteration one and two is different, iteration three is started. The two adjacent \textit{CNodes} are merged and result in the graph shown in figure B.9. No branches or loops are left, so the final graph is reached.

However, the algorithm recognises that in iteration three the node count was reduced, so a fourth iteration has to be executed. Since no changes in iteration four need to be made, the node count in iteration four is not less than the node count in iteration three, so the algorithm terminates. Since only one initial node, one final node and one \textit{CNode} remain, the postcondition is satisfied, and the execution of the algorithm was successful.

The remaining \textit{CNode} contains the \textit{C} code for the function body of a GCD. The resulting \textit{C} code is identical to the result of a manual implementation of Algorithm 2.
Figure 3.2.: Overview of the internal classes for activity diagrams
4. Evaluation and Validation

In this chapter, it will be shown that the algorithm described in the previous chapter terminates and that it works correctly. For the correctness directed and robustness tests are performed by implementing unit test and modelling input models for integration tests. The generated C file is validated through static code analysis against the MISRA-2012 standard. Finally, the challenges presented in section 2.4 are evaluated.

4.1. Algorithm Termination

To prove the termination of the algorithm, the compute function and every function, which is called by the compute function has to be proven to terminate. 'The termination proof requires a variant. A variant is an expression that is decremented with every iteration and has a lower bound. Once it has reached the lower bound, the loop has terminated.' [25, p. 361]. Alternatively, an upper bound and incrementing in each iteration are used to describe the variant. The loop respectively terminates when the upper bound is reached.

Assignments, branches and for-each loops do not prevent that the algorithm terminates. A for-each loop operates on a container with a fixed number of elements, which forms the upper bound. The incrementation of the variant happens implicitly by iterating over the elements. Since C# forbids adding or removing elements to the container inside of a for-each loop, these loops terminate if and only if their bodies terminate. Additionally, it is assumed in this thesis that external library calls always terminate. The function calls and other loops within the algorithm need to be checked for termination. The functions in the Graph class do not contain other function calls to functions within the project and do only use for-each loops. Thus, it can be stated that the replace, the combineCNodes, the Act2Graph and the DepthFirstSearch functions terminate. The compute function contains a while loop which terminates if the node count has not changed from the previous iteration. In each iteration before the last, the node count is reduced by at least one. Either the mergeCNodes() combines two adjacent nodes to one, mergeBranches() merges nodes to a single one if a branch exists, mergeLoops() replaces multiple nodes if a loop is detected, or the compute function terminates. An overview of the reasons for the termination of the complete compute function and its calls to other functions is given in table 4.1.

Other phenomena that could prevent termination, such as deadlocks or livelocks, cannot occur. This is the case because no concurrent computation takes place and the algorithm does not depend on any shared resources.

4.2. Algorithm Correctness

An algorithm is correct, if it does not contain errors, i.e. if it always produces the correct output for a given input, as long as certain preconditions are met. In other words,
### Function Name | Termination Reason
--- | ---
**compute** | lower bound for `graph.Vertices.Count()` is zero, either each iteration decrements or the loop terminates
**createCLocalVariables** | loop is for-each only contains external library calls
**DatastoreOrder** | loop is for-each only contains external library calls
**transformStructuredActivityNode2C** | loop is for-each recursively calls compute on subgraph (induction)
**transformStructuredNodeGraph2CNode** | loop is for-each recursively calls compute on subgraph (induction)
**transformAction2C** | loop is for-each only contains external library calls does not contain loops
**numerous transformation functions** | loop is for-each only contains external library calls does not contain loops
**replace** | loop is for-each only contains external library calls does not contain loops
**setupTimer** | loop is for-each only contains external library calls does not contain loops
**setupReceive** | loop is for-each only contains external library calls does not contain loops
**graph.DepthFirstSearch** | external library call
**mergeCNodes** | number of mergable nodes reduced by 1 every iteration, loop lower bound is 0 does not contain loops
**combineCNodes** | number of mergable nodes reduced by 1 every iteration, loop lower bound is 0 does not contain loops
**mergeBranches** | loop is bounded for-loop
**buildCSwitch** | loop is for-each
**buildCIf** | does not contain loops does not contain loops
**mergeLoops** | loop is bounded for-loop
**buildCLoop** | does not contain loops

Table 4.1.: Functions called by `compute` and reasons for their termination
4.2. Algorithm Correctness

Correctness of an implementation means consistency with the specification because the specification also decides what the correct output values are [17]. The term completeness is a special form of correctness and requires that everything required in the specification is realised.

To prove correctness, formal verification is required. In general, testing cannot be used to show the absence of errors; it can only show that the implementation produces the correct output for a given input or a range of inputs. This means that formal verification has the obvious advantage over testing that it is the only way to show that all possible behaviour of the implementation is correct. Testing, however, only "shows the presence of errors, not the absence' (Dijkstra, [25], slide 10).

Even though formal verification would be more expressive than testing, testing is used for validation in this thesis. This is due to the fact that the XMI input format is very diverse and complex, it is very difficult to find an expressive notation of transformation rules. These, however would be required to formally describe the specification. Since the specification cannot be formalised as part of this thesis, no formal proof of correctness is possible. Additionally, the sheer length and code complexity of the transformation algorithm and the use of external libraries put a formal verification far outside the scope of this thesis. To still remove as many faults as possible from the implementation, a variety of tests is used. These tests are described in the next subsections for unit tests (4.2.1) and for integration tests (4.2.2).

Due to the nature of the input format and due to the enforced design standards, some assumptions can be made that contribute towards correctness and that manifest as preconditions. Since if statements are not allowed without an else part, the dangling else problem cannot occur. It is established that most tools for designing UML models enforce the correct syntax [2] and it is assumed that they correctly export the models into XMI. Because no human is supposed to interfere with this representation, the precondition can be made that the input is syntactically correct. However, UML allows for ambiguities as stated by [2] and this can lead to unspecified behaviour.

4.2.1. Unit Testing

The unit tests used here only check for the correctness of the core algorithm itself. They operate directly on top of the internal graph representation. Thus, the front-end is not covered, hence no XMI file has to be loaded and parsed. The constructCFileContent function is also not covered because the UnitTestEvaluation function evaluates the output string of the algorithm directly against an expected string without printing the output into a file. Additionally, in the case of intentionally violated conditions, the test framework checks the type of the thrown exception and compares it to the expected type. A summary of all unit tests and their results can be found in appendix C. The report was generated automatically, thus an input graph cannot be generated for each case. The verdict of every unit test is successful because each test that has resulted in an error has been corrected. A System.Diagnostics.Contracts._ContractsRuntime.ContractException in the Actual Output column occurs if the condition of an assert within the algorithm is not satisfied.
### Table 4.2.: Transformation rules and the unit tests that cover them.

<table>
<thead>
<tr>
<th>Transformation Rule</th>
<th>Unit Tests</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1</td>
<td>12</td>
</tr>
<tr>
<td>A.2</td>
<td>12</td>
</tr>
<tr>
<td>A.3</td>
<td>only covered by integration tests</td>
</tr>
<tr>
<td>A.4</td>
<td>30</td>
</tr>
<tr>
<td>A.5</td>
<td>10</td>
</tr>
<tr>
<td>A.6</td>
<td>1,3,4,9,16,19,23,24,25,26,29,31,47,48,49,50,51</td>
</tr>
<tr>
<td>A.7</td>
<td>6</td>
</tr>
<tr>
<td>A.8</td>
<td>7</td>
</tr>
<tr>
<td>A.9</td>
<td>38,39,40</td>
</tr>
<tr>
<td>A.10</td>
<td>32,33,35</td>
</tr>
<tr>
<td>A.11</td>
<td>33,34,35</td>
</tr>
<tr>
<td>A.12</td>
<td>36,37</td>
</tr>
<tr>
<td>A.13</td>
<td>5,8,11,13,14,20,21,22,27,41,43,68</td>
</tr>
<tr>
<td>A.14</td>
<td>59,60,61,62,63,64,65</td>
</tr>
<tr>
<td>A.15</td>
<td>2,5,17,18,22,28,42,44,66,67</td>
</tr>
<tr>
<td>A.16</td>
<td>15,52,58</td>
</tr>
<tr>
<td>A.17</td>
<td>53,54,56</td>
</tr>
<tr>
<td>A.18</td>
<td>55,56,57</td>
</tr>
<tr>
<td>A.19</td>
<td>41,42,43,44,45,46</td>
</tr>
</tbody>
</table>

The unit tests are grouped by the function call in the `compute` function that they check. Each of these groups is implemented in a separate test suite, and they will be briefly introduced in the following paragraphs. For each test suite, both direct tests and robustness tests are performed. The test suites only check representatives of equivalence classes. Here, an equivalence class is defined as a set of inputs that are assumed to require the same steps to translate, based on the specification of the transformation rules. Each transformation rule is covered by at least one test, the mapping is given in table 4.2. Some rules are covered by multiple tests because different combinations of inputs are tested, and because different input parameters for the same rule produce different C-code. An example for this is an `if-then-else` construct that is described by a single rule, but the generated code depends on the exact number of `CNodes` and their placement in the graph.

The `UnitTestLocalVariables` test suite contains all tests belonging to the `createCLocalVariables` function. The different combinations of the `CLocalVariable` stereotype attributes are grouped into a smaller number of equivalence classes. To validate one representative for each equivalence class, the following test cases are included: 1, 3, 4, 9, 16, 19, 25, 31 and 47 - 51 (table in appendix C). Additionally, some test cases are designed to trigger exceptions that are caused by faulty inputs like test 23 and 26. Some test cases deliberately produce output that violates the MISRA coding rules in table
2.1, to check if these cases are handled (test 24 and 29).

The UnitTestStructuredNode test suite covers an activity with a structured node and its available stereotypes. Each structured node contains an internal graph, which will be transformed separately by a recursive call to the compute function. The selected internal graph will be checked separately, and each internal graph would result in the same equivalence class. Therefore, only an empty internal graph and an arbitrary, non-empty internal graph are tested in 15 and 52. According to rule 7 in table 2.1, data store nodes are not allowed within the internal graph, so an exception has to be thrown, and that is checked in test 58. Additionally, tests for each stereotype are performed. Test 53 to 57 contain one test with both stereotypes applied as well as a test with and without correct input for each stereotype.

The visit functions are tested in the UnitTestActActionNodes and UnitTestSignal test suites. For each action node, one test case with only this specific action node exists such as 6, 7, 10, 30, 32, 34, 36, 38. Here, the accept event action node is partitioned into blocking and non-blocking and a timer event. For sending and receiving a signal a stereotype is applied, so test cases that recognise a missing stereotype or a missing parameter need to be created. These test cases are 33, 37, 35, 39 and 40.

To show the correct concatenation of consecutive CNodes, a case, where two nodes are merged, is tested. A second test case covers the case that more than two CNodes need to be combined (test 45, 46). For one or zero CNodes, the mergeCNodes will not execute. This is confirmed by using an activity with only initial and final node as input as done in test 12.

The correct differentiation between an if-then-else and a switch-case is covered in the UnitTestBranch test suite. For an if-then-else it is tested in 11, 13, 20 and 21, that the output of an if-body, an else-body, none, or both is as expected and is compliant with the MISRA coding standard. Additionally, an exception need to be thrown in case that no condition for the if-then-else was provided (test 14) or if the decision node has only one outgoing edge (test 68). A nested if is checked in 27. A case like in 8 results in an exception, because a nested if with only one combined merge node is not allowed. For the switch-case a correct graph is tested in 59 with a CNode on the default path and in 62 with no node on the default path. Exceptional cases have none or more than one default path, a condition on the default edge, a missing stereotype or missing stereotype parameter. These possibilities are tested in cases 60, 61, 63, 64 and 65.

To validate the mergeLoops function, test case 2, 17 and 18 check cases with a correct input graph. Faulty inputs like missing CNodes or inserting the condition at the edge, which exits the loop, are covered in 66 and 67. Nested loops are tested in 28.

In UnitTestCombined each combination of a sequence of CNodes, branch and loop is tested. In test case 5 a branch has to be executed within a loop. Vice versa, a loop is executed in one path of a branch (test 22). A sequence in combination with a loop or a branch is tested in test 41 to 44.

With the unit tests a 100% line and branch coverage for the implementation of Algorithm 1 is reached. Due to debugging functionality in the graph class, the line and branch coverage do not reach 100%. The functions belonging to the front-end and the
4. Evaluation and Validation

one for the `constructCFileContent` function result in a coverage of zero, because they
are not addressed with the unit tests. However, these sections, as well as the `libXMI`,
will still be validated in the next subsection for integration testing. The code coverage
of the `UnitTestActivityGen` shows that each designed unit test is executed. An extract
of the code coverage report can be found in appendix D. The whole report can be found
on the attached CD.

4.2.2. Integration Testing

For the integration tests, three programs are modelled. The correctness of the trans-
formation is shown based on their output. With these tests, the correct parsing of
the elements into the internal structure and the correct writing into a C file should be
proven. The mapping itself has already been validated with the unit tests, so the three
test programs are sufficient to model each UML element once to show correct parsing
into the internal structure and writing into the output file. The call operation node
is only tested by unit tests, because in the XMI export of the PTC Integrity Modeler
an activity within an operation is not displayed in the XMI file. With the integration
tests, only directed tests are performed, since the robustness of the algorithm has been
shown with the unit tests. A mathematical problem is chosen as a test case because the
required output can be easily specified. The integration test should show that the data
flow in the model and in the C file are identical.

The GCD function is used to test the mapping of opaque action nodes, decision and
merge nodes to if-then-else and while loops. A modified version of the GCD has already
been explained in section 3.5. Additionally, the call behaviour node used in the cor-
responding main function is used to show the mapping to a correct function call with
the correct parameter modelled by the activity parameter node. Thus, testing a call
operation node designed in the PTC Integrity Modeler is not possible. The example
model calculates the GCD of 4 and 2, which is expected to be 2. The code generator
produces the `ggt_neu.c`, which is compiled with the `gcc` and prints the output to the
console, which is 2 as expected. Since the produced C code was manually compared to
the reference implementation in algorithm 2 and since it can be compiled without errors,
no further input value pairs have to be tested. The input model and the internal graph
structure can be found in appendix E.1. The resulting C file is in appendix F.1.

In the second test, a simple calculator, is modelled to map the send signal and the
accept event action node to the OAL functions for sending a message and a blocking
receive. Additionally, data store nodes are mapped to variable declarations. The cre-
ateObjectAction node is not tested because it should not be used. In this example also
a switch-case is modelled with opaque action nodes, a decision and a merge node. Com-
ilation with a native gcc and execution on the host system are not possible for the
calculator.c because the OAL functions call into an RTOS. Thus, the source code file is
only reviewed, but not executed. The input model and the internal graph structure can
be found in appendix E.2. The correspondent C file is shown in appendix F.3.

The third test category tests the structured nodes. The input model and the internal
graph structure can be found in appendix E.3. The resulting C file can be found in
appendix F.4. Therefore, a *structured node*, a structured node with the *for-loop* stereotype and a *structured node* with both stereotypes (*non-interruptible region* and *for-loop*) are applied. Each *structured node* contains a simple graph with *initial node*, *final node* and *opaque action node* since it is not important what is modelled within the *structured node* itself due to the recursive processing of the graph within the *structured node*. As a result, the `resultIteration3.c` is generated. The source code file is only reviewed, but not executed, because also OAL functions are used, and no mathematical problem was chosen as the basis of this example. todowhy? For the *fork* and *join node* only a concept was developed, so these nodes are not tested.

To check for a correct mapping from the model into the internal graph, a *graphML* representation of this internal data structure is created for each test. One *graphML* file is created before the algorithm starts, but after the XMI file has been parsed. This *graphML* file is compared manually with the original model designed in the PTC Integrity Modeler. For this, the *graphML* files are displayed with *yEd* [31] version 3.15.0.2. The results can be found in appendix E.

### 4.2.3. Static Code Analysis

To check whether the generated output file is compliant to the MISRA coding standard [18], the static code analysis tool *Understand* is used [26]. The *C* output files from the test cases of the integration tests are checked with this static code analysis. There are three categories where non-compliant source code can be produced. The first category is the mapping of the code generator creates non-compliant source code. The second category results from the design the user has modelled like two points of exit, non-compliant source code in *opaque action nodes*, or lacking design possibilities like the return value name or specifying signatures. The third category deals with changes after the code generation, like including a standard library. To avoid non-compliant source code by the designer, the DO-331 defines a modelling standard. In the future, a model checker will be implemented to indicate these violations early. The issues found with the static code analysis are described in the following.

To be able to print the results of the GCD implementation, I/O functions from the *C* standard library are used, and the library has been included manually. According to MISRA, standard libraries should not be used. Since the include-directive was added manually after the synthesis, this does not indicate an error in the code generator.

The PTC Integrity Modeler, unfortunately, does not offer to name pins used for accessing return values. Instead, these always take the name of the called function which means that e.g. the result of a call to the function *ggt* is stored in the variable *ggt* (listing F.1, line 31). This leads to a violation of the MISRA standard that demands identifiers to be unique. Solutions are already presented in section 3.3, when explaining the *transformAction2C* function. So the user can solve it by changing the design.

For a better comparability, the model of the GCD is designed according to the reference implementation (algorithm 2). This means, two points of exit are modelled, which results in a non-compliant source code as pointed out by the static code analysis. Since design decisions like this are transferred from the model to the resulting source code, it is
the responsibility of the designer to create a compliant model. A version of the GCD program that was created from a model with a single point of exit is given in listing F.2.

Activity diagrams are meant to provide only the body of functions or methods. To be able to test the results of the code generator developed in this thesis, function signatures are generated as well. However, since the purpose of operation diagrams is to provide complete function signatures, and since activity diagrams cannot specify all modifiers needed, this thesis will only add minimal function signatures needed for testing. This means that the MISRA issues connected to function signatures can be safely ignored. In addition, the release structure of generated programs is addressed by a separate thesis and are not part of the scope here. Hence, no header files have been included in the static code analysis as the sole point was to verify the generated function bodies. This results in a variety of issues listed by the analysis tool, that are connected to non-existent function declarations and definitions.

As discussed above, none of the listed violations of the MISRA-2012 rules is caused by errors in the implemented code generator. In all cases, the designed model was non-compliant to begin with, the analysis tool could not cope with lacking infrastructure such as missing header files with declarations, or manual changes are performed after the synthesis. Thus, the implementation of the code generator has passed the static code analysis. The complete MISRA-2012 report can be found in appendix F.

4.3. Evaluation

The challenges for this thesis described in section 2.4 are evaluated in this section. Showing the correctness and termination of the algorithm has already been done in the sections above.

Another challenge is to avoid dead code. Since no fixed design pattern is used and only modelled elements are transformed, dead code is not generated.

An additional challenge is a documented and traceable code, which can be achieved by modelling requirements and comments. However, in the PTC Integrity Modeler version 8.2.15, the XMI export for requirements and comments is faulty, so code generation for these elements is not possible at the moment. The readability of the code is accomplished by the use of GNUIndent [5] which is invoked after storing the source code in a file.

The challenge to distinguish between branches and loops and to identify them was solved by applying the depth-first search algorithm to the internal graph, which colours the edges.

One challenge results from the possible dependencies between the data store nodes, which are modelled by data flow edges. The instructions are scheduled according to the result of the topological sort and are added at the beginning of a function.

Another challenge arises due to the fact that abstract concepts like parallelism can be easily expressed with UML elements, but not with the C language. So operating system functions, which are specified in the OAL, are used to bridge the semantic gap.

This semantic gap between the UML and the C language can also be bridged by applying stereotypes. With these, missing information can be specified for example
whether an accept event action is blocking or non-blocking. Because the code generator enforces the specification, the intention of the user is clear. A manual implementer would sometimes try to guess missing information. It is possible that the implementer makes mistakes during this step and this can be avoided by automation.

Some stereotypes like the CLocalVariable require certain input data such as the default value. Stereotypes do not offer the possibility to add pins with a fixed name to action nodes when creating them with the GUI. Adding pins manually by each user could result in undefined attribute names and an automatic association to the OAL function arguments becomes difficult. The association between the parameters in a function signature with the ones in the function call is handled by sorting the parameters by their names, but therefore the names need to be equal. Thus, the user can publish the information such as the default value in a separate attribute window with a specified name after applying the stereotype. Unfortunately, this interrupts the data flow, and the input data cannot be checked automatically for their data type. To design stereotypes with named pins, the stereotype has to be created by using a visual basic script, which adds pins automatically with the right name and type to the UML element. This information was not available during the implementation phase of this thesis, so the stereotypes are used with the separate attribute window.

When using pins, another problem occurs. If an opaque action node modifies a variable, an input pin with the same name as an output pin would be required to track the data flow correctly, but action nodes cannot have an input pin and an output pin with the same name. As a solution, I/O flow edges can be used instead of the data flow edges. The I/O flow was introduced after finishing the implementation phase, so it is not implemented in the code generator. The I/O flow edge carries information about the variable name and data type. Thus, identifying and tracking the data flow is independent of the name of the input and output pin. With the I/O flow, the data flow can also be tracked in opaque action nodes, so the data flow in the model can be compared with the data flow in the output C files. However, variables that are used inside guards for branch or loop conditions still cannot be tracked without analysing the stored string.

To conclude, this thesis has solved the main challenges of code generation in concept and states the topics, in which further investigation is needed like including I/O flow edges for data flow challenges and requirements.
5. Conclusion

A commercial code generator for activity diagrams that fulfils the claimed requirements of avionic systems has not been implemented, yet. So the aim of this thesis is to work out a transformation, which takes UML activity diagrams as input and automatically translates them into ANSI-C source code, which conforms to the MISRA standard. The code generator helps to find errors early in the development life-cycle when the costs for an error are low. Additionally, misunderstandings can be identified easily, due to the comprehensible representation of models.

For the transformation, the elements occurring in an activity diagram are identified, and a suitable mapping is presented. To implement this mapping, an algorithm is developed, which succeed in implementing the transformation so that the code generator can be used in the context of avionic systems. The algorithm faces the challenge of closing the semantic gap between the UML and the C language. To archive this, functions from an underlying operating system, which is abstracted by the Operation System Abstraction Layer (OAL), is used as well as stereotypes that add the missing information to the UML elements. Additionally, algorithms from the field of graph theory are used to identify branches and loops in the activity graph and to manage the order of variable definitions. Finally, the implemented algorithm is validated by showing termination and correctness. For the correctness directed and robustness unit tests and direct integration tests are performed as well as static code analysis on the resulting C file.

In conclusion, a satisfying solution has been found for the control flow challenges, while the data flow challenges require further investigations.

5.1. Outlook

In the future work a code generator should be developed that can handle all UML/SysML diagrams. Furthermore, the code generator needs to be certified according to the DO-330, so the V-model becomes a Y-model as intended in the motivation. Thus, the validation has to be done only on the model and not anymore on the source files. Additionally, the aim is to simulate the model and to mark the control and data flow edges which are covered by tests, to visualise the coverage. The future plan is to have a model checker, which ensures that every input fulfils the preconditions before starting the algorithm.

To complete the activity diagram code generator, the transformation of I/O flow edges need to be supported to model and track the data flow in the model correctly. In addition, requirements have to be transformed. Due to the faulty XMI export, this has not been done, yet. To solve this problem, another approach is considered. The model data is caught directly from the PTC Integrity Modeler repository by using the automation interface instead of the XMI export. The advantage is that all model data is available in contrast to the XMI file where the export is not complete like for modelling requirements or the call operation action node. On the other hand, the tool independence can get lost.
A. Transformation Rules

UML initial node $\Rightarrow \emptyset$ \hspace{1cm} (A.1)

UML activity final node $\Rightarrow \emptyset$ \hspace{1cm} (A.2)

UML activity $\times$ UML activity parameter node $\Rightarrow$ C function signature \hspace{1cm} (A.3)

UML opaque action node $\Rightarrow$ C statements \hspace{1cm} (A.4)

UML create object action node $\times$ UML output pin $\times$ UML primitive type $\Rightarrow$ C local variable \hspace{1cm} (A.5)

UML data store node $\times$ CLocalVariable stereotype $\Rightarrow$ (C local variable$\mid$C pointer variable$\mid$C array$\mid$C bitfield) $\times$ default value \hspace{1cm} (A.6)

UML call behaviour action node $\times$ UML input pin $\times$ UML output pin $\Rightarrow$ C function call \hspace{1cm} (A.7)

UML call operation action node $\times$ UML input pin $\times$ UML output pin $\Rightarrow$ C function call \hspace{1cm} (A.8)

UML send signal action node $\times$ SndSig stereotype $\Rightarrow$ OAL_MsgSendViaId function call \hspace{1cm} (A.9)

UML accept event action node $\times$ RcvEvent stereotype $\times$ blocking $\Rightarrow$ OAL_MsgRecvBlk function call \hspace{1cm} (A.10)
UML accept event action node \times \text{RcvEvent stereotype} \times \text{non blocking} \\
\Rightarrow \text{OAL\_MsgRecvNonBlk function call} \quad (A.11)

UML accept event action node \times \text{RcvEvent stereotype} \times \text{UmlActTimeEvent} \\
\Rightarrow \text{OAL\_SetTimer function call} \quad (A.12)

UML decision node \times \text{UML merge node} \times \text{CNode} \times \text{UML control flow edges} \times \text{guards} \times \text{one forward-cross edge} \\
\Rightarrow \text{C if-then-else} \quad (A.13)

UML decision node \times \text{UML merge node} \times \text{CNode} \times \text{UML control flow edges} \times \text{guards} \times \text{forward-cross edges} \times \text{switch stereotype} \\
\Rightarrow \text{C switch case} \quad (A.14)

UML decision node \times \text{UML merge node} \times \text{CNode} \times \text{UML control flow edges} \times \text{guards} \times \text{one back edge} \\
\Rightarrow \text{C while loop/C do-while loop} \quad (A.15)

UML structured node \times \text{UML activity graph in UML structured node} \\
\Rightarrow \text{C code for UML activity graph} \quad (A.16)

UML structured node \times \text{CForLoop stereotype} \times \\
\text{UML activity graph in UML structured node} \\
\Rightarrow \text{C for loop} \times \text{C code for UML activity graph} \quad (A.17)

UML structured node \times \text{NonInterrupibleRegion stereotype} \times \\
\text{UML activity graph in UML structured node} \\
\Rightarrow \text{enabling/disabling interrupts} \times \text{C code for UML activity graph} \quad (A.18)
The following rules only have been created conceptually. For various reasons they are not part of the implementation.

- UML fork node $\Rightarrow$ OAL\_TaskSpawn function call
- UML join node $\Rightarrow$ synchronisation with semaphores or event flags
- UML expansion region $\Rightarrow$ C for loop with arrays
- UML interruptible region $\Rightarrow$ Exceptions in object oriented target language
B. Greatest Common Divisor (gcd)

```c
int a = 10; int b = 15;
return b;
return a;
a = a - b; b = b - a;
```

Figure B.1.: gcd internal graph
B. Greatest Common Divisor (gcd)

```c
int a = 10;
int b = 15;

if (a == 0) return b;
else if (b != 0) {  // Loop
    a = a - b;
    b = b - a;
    if (a > b) goto start;
    else goto start2;
}

return a;
```

Figure B.2.: gcd internal graph after `createCLocalVariables`
Figure B.3.: gcd internal graph after transformAction2C
Figure B.4.: gcd internal graph after applying the depth first search
int a = 10;  
int b = 15;

if (a > b)  
{  
a = a - b;
}
else  
{
    b = b - a;
}

Figure B.5.: gcd internal graph after mergeBranches in iteration 1
B. Greatest Common Divisor (gcd)

```c
int a = 18;
int b = 15;

while(b != 0)
{
    if(a > b)
    {
        a = a - b;
    }
    else
    {
        b = b - a;
    }
}
return a;
```

Figure B.6.: gcd internal graph after mergeLoops in iteration 1
int a = 10;
int b = 15;

while(b != 0)
{
    if(a > b)
    {
        a = a - b;
    }
    else
    {
        b = b - a;
    }
}

return a;

[a==0]
return b;

Figure B.7.: gcd internal graph after mergeCNodes in iteration 2
B. Greatest Common Divisor (gcd)

```c
int a = 10;
int b = 15;
if(a == 0)
{
    return b;
}
else
{
    while(b != 0)
    {
        if(a > b)
        {
            a = a - b;
        }
        else
        {
            b = b - a;
        }
    }
    return a;
}
```

Figure B.8.: gcd internal graph after `mergeBranches` in iteration 2
Algorithm 2 Greatest Common Divisor (Euclid’s original algorithm as described in [16])

```
function \text{GCD} \n\quad \text{if } a = 0 \text{ then} \n\quad \quad \text{return } b \n\quad \text{while } b \neq 0 \text{ do} \n\quad \quad \text{if } a > b \text{ then} \n\quad \quad \quad a \leftarrow a - b \n\quad \quad \text{else} \n\quad \quad \quad b \leftarrow b - a \n\quad \quad \text{return } a
```

Figure B.9.: gcd internal graph after \texttt{mergeCNodes} in iteration 3

```
int a = 10;
int b = 15;

if(a == 0)
{
    return b;
}
else
{
    while(b != 0)
    {
        if(a > b)
        {
            a = a - b;
        }
        else
        {
            b = b - a;
        }
    }
    return a;
}
```
## C. Test Cases

<table>
<thead>
<tr>
<th>No</th>
<th>Test Name</th>
<th>Verdict</th>
<th>Expected Output</th>
<th>Actual Output</th>
<th>Input Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>TestArray</td>
<td>successful</td>
<td>int8_t i[2] = { 0,1 };</td>
<td>int8_t i[2] = { 0,1 };</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Test-Backward-LoopBody</td>
<td>successful</td>
<td>while( x&lt; = 10 ) { A1; }</td>
<td>while( x&lt; = 10 ) { A1; }</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>TestBasic-Type</td>
<td>successful</td>
<td>int8_t i = 0;</td>
<td>int8_t i = 0;</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Test-Bitfield</td>
<td>successful</td>
<td>struct{ int8_t x:3, y:5; } point = { 1,16 };</td>
<td>struct{ int8_t x:3, y:5; } point = { 1,16 };</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Test-BranchIn-Loop</td>
<td>successful</td>
<td>while( x&lt; = 10 ) { if( y! = 10 ) { A1; } else { A2; } }</td>
<td>while( x&lt; = 10 ) { if( y! = 10 ) { A1; } else { A2; } }</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>TestCall-Behaviour-Node</td>
<td>successful</td>
<td>double callTest Function = callTest Function(a,b,&amp;c);</td>
<td>double callTest Function = callTest Function(a,b,&amp;c);</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>TestCall-Operation-Node</td>
<td>successful</td>
<td>double callTest Function = callTest Function(a,b,&amp;c);</td>
<td>double callTest Function = callTest Function(a,b,&amp;c);</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>TestConst</td>
<td>successful</td>
<td>const int8_t i = 0;</td>
<td>const int8_t i = 0;</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Test-Create-ObjNode</td>
<td>successful</td>
<td>int i;</td>
<td>int i;</td>
<td></td>
</tr>
</tbody>
</table>
| Test Case | Status | Description | Exception | Activity
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>11 TestElse-BODY</td>
<td>successful</td>
<td>Exception thrown</td>
<td>ActivityGen. Exception.NoIf BodyException</td>
<td></td>
</tr>
<tr>
<td>12 Test-EMPTY</td>
<td>successful</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13 Test-EMPTY-BODY</td>
<td>successful</td>
<td>Exception thrown</td>
<td>ActivityGen. Exception.NoIf BodyException</td>
<td></td>
</tr>
<tr>
<td>14 Test-EMPTY-CONDITION</td>
<td>successful</td>
<td>Exception thrown</td>
<td>ActivityGen. Exception.Illegal Condition</td>
<td></td>
</tr>
<tr>
<td>15 Test-EMPTY-STRUCTURE-NODE</td>
<td>successful</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16 Test-EXTERN</td>
<td>successful</td>
<td>extern int8_t i;</td>
<td>extern int8_t i;</td>
<td></td>
</tr>
<tr>
<td>17 Test-FORWARD-AND-BACKWARD-LOOPBODY</td>
<td>successful</td>
<td>A1; while( x&lt; = 10 ) { A2; A1; }</td>
<td>A1; while( x&lt; = 10 ) { A2; A1; }</td>
<td></td>
</tr>
<tr>
<td>18 Test-FORWARD-LOOPBODY</td>
<td>successful</td>
<td>do { A1; } while( x&lt; = 10 );</td>
<td>do { A1; } while( x&lt; = 10 );</td>
<td></td>
</tr>
<tr>
<td>19 Test-FUNCTION-POINTER</td>
<td>successful</td>
<td>Exception thrown</td>
<td>ActivityGen. Exception.No FunctionPointer Allowed</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Test-Case</td>
<td>Success</td>
<td>Code</td>
<td>Code</td>
</tr>
<tr>
<td>---</td>
<td>----------------</td>
<td>---------</td>
<td>----------------------------------------------------------------------</td>
<td>----------------------------------------------------------------------</td>
</tr>
<tr>
<td>20</td>
<td>TestIf-AndElse-Body</td>
<td>successful</td>
<td>if ( x&lt; = 10 ) { A1; } else { A2; }</td>
<td>if ( x&lt; = 10 ) { A1; } else { A2; }</td>
</tr>
<tr>
<td>21</td>
<td>TestIf-Body</td>
<td>successful</td>
<td>if ( x&lt; = 10 ) { A1; } else { NOOP; }</td>
<td>if ( x&lt; = 10 ) { A1; } else { NOOP; }</td>
</tr>
<tr>
<td>22</td>
<td>TestLoop-InBranch</td>
<td>successful</td>
<td>if ( x&lt; = 10 ) { while ( y! = 10 ) { A1; } else { NOOP; } }</td>
<td>if ( x&lt; = 10 ) { while ( y! = 10 ) { A1; } else { NOOP; } }</td>
</tr>
<tr>
<td>23</td>
<td>Test-Missing-Stereotype</td>
<td>successful</td>
<td>Exception thrown</td>
<td>ActivityGen. Exception.NoC VariableStereotype Applied</td>
</tr>
<tr>
<td>24</td>
<td>Test-Multiple-Assignment</td>
<td>successful</td>
<td>Exception thrown</td>
<td>ActivityGen. Exception.No MultipleAssignments</td>
</tr>
<tr>
<td>25</td>
<td>Test-Multiple-Variables</td>
<td>successful</td>
<td>int8_t i = 0; int8_t *p = &amp;i; int8_t *q = p;</td>
<td>int8_t i = 0; int8_t *p = &amp;i; int8_t *q = p;</td>
</tr>
<tr>
<td>26</td>
<td>Test-Multiple-Variables-With-Loop-Objectflow</td>
<td>successful</td>
<td>Exception thrown</td>
<td>QuickGraph.Non AcyclicGraph Exception</td>
</tr>
<tr>
<td>27</td>
<td>Test-NestedIf</td>
<td>successful</td>
<td>if ( x&lt; = 10 ) { if ( y! = 10 ) { A1; } else { A2; } else { NOOP; } }</td>
<td>if ( x&lt; = 10 ) { if ( y! = 10 ) { A1; } else { A2; } else { NOOP; } }</td>
</tr>
</tbody>
</table>
## C. Test Cases

<table>
<thead>
<tr>
<th>Test Case</th>
<th>Description</th>
<th>Status</th>
<th>Code Snippet 1</th>
<th>Code Snippet 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>Test Nested-Loop</td>
<td>successful</td>
<td>(A1; \text{while}(x \leq 10)) { \text{while}(y! = 10) { A2; } A1; } }</td>
<td>(A1; \text{while}(x \leq 10)) { \text{while}(y! = 10) { A2; } A1; } }</td>
</tr>
<tr>
<td>29</td>
<td>Test Not-Initialised-Variable</td>
<td>successful</td>
<td>Exception thrown</td>
<td>(\text{ActivityGen}.\text{Exception.Variable IsNotInitialisedAt Declaration})</td>
</tr>
<tr>
<td>30</td>
<td>Test Opaque-Node</td>
<td>successful</td>
<td>int (i = 0; ) \text{while}(i&lt;10){ \text{int }i = 0; \text{while}(i&lt;10){ i++; } }</td>
<td>(\text{int }i = 0; \text{while}(i&lt;10){ i++; } )</td>
</tr>
<tr>
<td>31</td>
<td>Test Pointer</td>
<td>successful</td>
<td>(\text{int8_t }*p = \text{null};)</td>
<td>(\text{int8_t }*p = \text{null};)</td>
</tr>
<tr>
<td>32</td>
<td>Test Receive-SignalBlk</td>
<td>successful</td>
<td>(\text{ret }= \text{OAL_MsgRecvBlk(1, &amp;buf, 16, 1, 2);})</td>
<td>(\text{ret }= \text{OAL_MsgRecvBlk(1, &amp;buf, 16, 1, 2);})</td>
</tr>
<tr>
<td>33</td>
<td>Test Receive-Signal-Missing-Parameter</td>
<td>successful</td>
<td>Exception thrown</td>
<td>(\text{ActivityGen}.\text{Exception. StereotypeProperty Missing})</td>
</tr>
<tr>
<td>34</td>
<td>Test Receive-Signal-NonBlk</td>
<td>successful</td>
<td>(\text{ret }= \text{OAL_MsgRecvNonBlk(1, &amp;buf, 16, 1);})</td>
<td>(\text{ret }= \text{OAL_MsgRecvNonBlk(1, &amp;buf, 16, 1);})</td>
</tr>
<tr>
<td>35</td>
<td>Test Receive-Signal-Without-Stereotype</td>
<td>successful</td>
<td>Exception thrown</td>
<td>(\text{ActivityGen}.\text{Exception.NoSignal StereotypeApplied})</td>
</tr>
<tr>
<td>36</td>
<td>Test Receive-Timer-Signal</td>
<td>successful</td>
<td>(\text{timer }= \text{OAL_TimerStart(timer Msg, 2);} \text{ret }= \text{OAL_MsgRecvNonBlk(1, &amp;buf, 16, 1);})</td>
<td>(\text{timer }= \text{OAL_TimerStart(timer Msg, 2);} \text{ret }= \text{OAL_MsgRecvNonBlk(1, &amp;buf, 16, 1);})</td>
</tr>
<tr>
<td>37</td>
<td>Test Receive-Timer-Signal-Missing-Parameter</td>
<td>successful</td>
<td>Exception thrown</td>
<td>(\text{ActivityGen}.\text{Exception. StereotypeProperty Missing})</td>
</tr>
<tr>
<td></td>
<td>TestSend-Signal</td>
<td>successful</td>
<td>int32_t err = OAL_MsgSendViaId(1, 1, &amp;msg, 16, OAL_pri Normal);</td>
<td>TestSend-Signal successful</td>
</tr>
<tr>
<td>---</td>
<td>----------------</td>
<td>------------</td>
<td>----------------------------------------------------------------</td>
<td>-----------------------------</td>
</tr>
<tr>
<td>38</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>39</td>
<td>TestSend-Signal-Missing-Parameter</td>
<td>successful</td>
<td>Exception thrown</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>TestSend-Signal-Without-Stereotype</td>
<td>successful</td>
<td>Exception thrown</td>
<td></td>
</tr>
<tr>
<td>41</td>
<td>Test-Sequence-NodeAnd-Branch</td>
<td>successful</td>
<td>A1; if( x&lt; = 10 ) { A2; } else { A3; }</td>
<td></td>
</tr>
<tr>
<td>42</td>
<td>Test-Sequence-NodeAnd-Loop</td>
<td>successful</td>
<td>A1; while( x&lt; = 10 ) { A2; }</td>
<td></td>
</tr>
<tr>
<td>43</td>
<td>Test-Sequence-NodeIn-Branch</td>
<td>successful</td>
<td>if( x&lt; = 10 ) { A1; A2; } else { A3; }</td>
<td></td>
</tr>
<tr>
<td>44</td>
<td>Test-Sequence-NodeIn-Loop</td>
<td>successful</td>
<td>while( x&lt; = 10 ) { A1; A2; }</td>
<td></td>
</tr>
<tr>
<td>Test Case</td>
<td>Successful</td>
<td>Code 1</td>
<td>Code 2</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>------------</td>
<td>--------</td>
<td>--------</td>
<td></td>
</tr>
<tr>
<td>C. Test Cases</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Test-Sequence-OfThree</td>
<td>successful</td>
<td>A1; A2; A3;</td>
<td>A1; A2; A3;</td>
<td></td>
</tr>
<tr>
<td>Test-Sequence-OfTwo</td>
<td>successful</td>
<td>A1; A2;</td>
<td>A1; A2;</td>
<td></td>
</tr>
<tr>
<td>TestStatic</td>
<td>successful</td>
<td>static int8_t i = 0;</td>
<td>static int8_t i = 0;</td>
<td></td>
</tr>
<tr>
<td>Test-Static-Const-Array</td>
<td>successful</td>
<td>static const int8_t i[2] = {0, 1};</td>
<td>static const int8_t i[2] = {0, 1};</td>
<td></td>
</tr>
<tr>
<td>Test-Static-Const-Bitfield</td>
<td>successful</td>
<td>static const struct {int8_t x:3, y:5; } point = {1,16};</td>
<td>static const struct {int8_t x:3, y:5; } point = {1,16};</td>
<td></td>
</tr>
<tr>
<td>Test-Static-Const-Pointer</td>
<td>successful</td>
<td>static const int8_t *i = 0;</td>
<td>static const int8_t *i = 0;</td>
<td></td>
</tr>
<tr>
<td>Test-Static-Const-Variable</td>
<td>successful</td>
<td>static const int8_t i = 0;</td>
<td>static const int8_t i = 0;</td>
<td></td>
</tr>
<tr>
<td>Test-Structure-Node</td>
<td>successful</td>
<td>A1;</td>
<td>A1;</td>
<td></td>
</tr>
<tr>
<td>Test-Structure-NodeFor-Loop</td>
<td>successful</td>
<td>for(i = 0; i&lt;10; i++){A1; }</td>
<td>for(i = 0; i&lt;10; i++){A1; }</td>
<td></td>
</tr>
<tr>
<td>Test-Structure-Node-ForLoop-Missing-Parameter</td>
<td>successful</td>
<td>Exception thrown</td>
<td>ActivityGen .Exception, StereotypeProperty Missing</td>
<td></td>
</tr>
<tr>
<td>Test-Structure-NodeNon-Interruptable</td>
<td>successful</td>
<td>var = OAL_Int Lock(); A1; OAL_IntUnlock(var);</td>
<td>var = OAL_Int Lock(); A1; OAL_IntUnlock(var);</td>
<td></td>
</tr>
<tr>
<td>Step</td>
<td>Test Case Description</td>
<td>Diagram</td>
<td>Code Snippet 1</td>
<td>Code Snippet 2</td>
</tr>
<tr>
<td>------</td>
<td>----------------------</td>
<td>---------</td>
<td>----------------</td>
<td>----------------</td>
</tr>
<tr>
<td>56</td>
<td>Test-Structure-NodeNon-InterruptableAnd-ForLoop</td>
<td><img src="image1.png" alt="Diagram" /></td>
<td><code>var = OAL_Int Lock(); for(i = 0; i&lt;10; i++){ A1; A2; } OAL_Int Unlock(var);</code></td>
<td><code>var = OAL_Int Lock(); for(i = 0; i&lt;10; i++){ A1; A2; } OAL_Int Unlock(var);</code></td>
</tr>
<tr>
<td>57</td>
<td>Test-Structure-NodeNon-Interruptable-Missing-Parameter</td>
<td><img src="image2.png" alt="Diagram" /></td>
<td></td>
<td></td>
</tr>
<tr>
<td>58</td>
<td>Test-Structure-Nodewith-DataStore</td>
<td><img src="image3.png" alt="Diagram" /></td>
<td></td>
<td></td>
</tr>
<tr>
<td>59</td>
<td>Test-Switch</td>
<td><img src="image4.png" alt="Diagram" /></td>
<td><code>switch( a ){ case 1: A1; break; case 2: break; default: A3; break; }</code></td>
<td><code>switch( a ){ case 1: A1; break; case 2: break; default: A3; break; }</code></td>
</tr>
<tr>
<td>60</td>
<td>Test-Switch2-Default-Paths</td>
<td><img src="image5.png" alt="Diagram" /></td>
<td></td>
<td></td>
</tr>
<tr>
<td>61</td>
<td>Test-Switch-GuardOn-Default-Path</td>
<td><img src="image6.png" alt="Diagram" /></td>
<td></td>
<td></td>
</tr>
<tr>
<td>62</td>
<td>Test-Switch-With-Default-Path</td>
<td><img src="image7.png" alt="Diagram" /></td>
<td><code>switch( a ){ case 1: A1; break; case 2: A3; break; default: break; }</code></td>
<td><code>switch( a ){ case 1: A1; break; case 2: A3; break; default: break; }</code></td>
</tr>
<tr>
<td>Test Case</td>
<td>Status</td>
<td>Exception Thrown</td>
<td>ActivityGen. Exceptions</td>
<td></td>
</tr>
<tr>
<td>--------------------------------------------------------------------------</td>
<td>---------</td>
<td>---------------------------------------</td>
<td>-------------------------</td>
<td></td>
</tr>
<tr>
<td>Test-Switch-Without-Default-Edge</td>
<td>successful</td>
<td>Exception thrown</td>
<td>Exception.IllegalDefaultPathNumber</td>
<td></td>
</tr>
<tr>
<td>Test-Switch-Without-Stereotype</td>
<td>successful</td>
<td>Exception thrown</td>
<td>Exception.NoSwitchStereotypeApplied</td>
<td></td>
</tr>
<tr>
<td>Test-Switch-Without-Stereotype-Property</td>
<td>successful</td>
<td>Exception thrown</td>
<td>Exception.StereotypePropertyMissing</td>
<td></td>
</tr>
<tr>
<td>TestWith-Guard-OnOther-Edge</td>
<td>successful</td>
<td>Exception thrown</td>
<td>Exception.IllegalCondition</td>
<td></td>
</tr>
<tr>
<td>Test-Without-Body</td>
<td>successful</td>
<td>Exception thrown</td>
<td>Exception.NoLoopBodyException</td>
<td></td>
</tr>
</tbody>
</table>
D. Test Coverage

Generated on: 26.09.2016 - 08:23:30
Parser: OpenCoverParser
Assemblies: 3
Classes: 103
Files: 81
Covered lines: 2481
Uncovered lines: 2203
Coverable lines: 4684
Total lines: 9381
Line coverage: 52.9%
Branch coverage: 22.8%

D.1. Assemblies

ActivityGen  49%
ActivityGen.ActGen  0%
ActivityGen.Algorithm  100%
ActivityGen.CNode  84.2%
ActivityGen.Graph  83.3%
ActivityGen.Loop  100%
ActivityGen.PostTranslationActions  0%
ActivityGen.Program  0%
ActivityGen.TranslationActions  0%
ActivityGen.UmlEdge  100%
System.Diagnostics.Contracts.RuntimeContractsAttribute  0%
libXMI  17%
libXML.Log  24%
libXMI.SysML.Structure.SysMLBlock  0%
libXMI.UML.Edges.UmlControlFlow  70%
libXMI.UML.Edges.UmlFlow  41.1%
libXMI.UML.Edges.UmlObjectFlow  93.7%
libXMI.UML.Events.UmlActTimeEvent  8.3%
libXMI.UML.Events.UmlCallEvent  0%
libXMI.UML.Events.UmlEvent  6.6%
libXMI.UML.Events.UmlSignal  0%
libXMI.UML.Events.UmlSignalEvent  7.1%
libXMI.UML.Events.UmlTimeEvent  0%
libXMI.UML.Events.UmlTimeExpression  0%
libXMI.UML.Events.UmlTrigger  38.8%
libXMI.UML.Nodes.UmlAcceptEventAction  77.7%
### D. Test Coverage

<table>
<thead>
<tr>
<th>Class/Node</th>
<th>Coverage (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>libXMI.UML.Nodes.UmlActAction</td>
<td>16.6%</td>
</tr>
<tr>
<td>libXMI.UML.Nodes.UmlActivityFinalNode</td>
<td>17.3%</td>
</tr>
<tr>
<td>libXMI.UML.Nodes.UmlActivityParameterNode</td>
<td>0%</td>
</tr>
<tr>
<td>libXMI.UML.Nodes.UmlBranchFlowNode</td>
<td>8.8%</td>
</tr>
<tr>
<td>libXMI.UML.Nodes.UmlCallBehaviourAction</td>
<td>45%</td>
</tr>
<tr>
<td>libXMI.UML.Nodes.UmlCallOperationAction</td>
<td>45%</td>
</tr>
<tr>
<td>libXMI.UML.Nodes.UmlCreateObjectAction</td>
<td>41.1%</td>
</tr>
<tr>
<td>libXMI.UML.Nodes.UmlDataStoreNode</td>
<td>24.5%</td>
</tr>
<tr>
<td>libXMI.UML.Nodes.UmlDecisionNode</td>
<td>81.2%</td>
</tr>
<tr>
<td>libXMI.UML.Nodes.UmlForkNode</td>
<td>0%</td>
</tr>
<tr>
<td>libXMI.UML.Nodes.UmlInitialNode</td>
<td>16%</td>
</tr>
<tr>
<td>libXMI.UML.Nodes.UmlInputPin</td>
<td>6.6%</td>
</tr>
<tr>
<td>libXMI.UML.Nodes.UmlJoinBranchFlowNode</td>
<td>8.8%</td>
</tr>
<tr>
<td>libXMI.UML.Nodes.UmlJoinNode</td>
<td>0%</td>
</tr>
<tr>
<td>libXMI.UML.Nodes.UmlMergeNode</td>
<td>80%</td>
</tr>
<tr>
<td>libXMI.UML.Nodes.UmlOpaqueAction</td>
<td>47%</td>
</tr>
<tr>
<td>libXMI.UML.Nodes.UmlOutputPin</td>
<td>6.6%</td>
</tr>
<tr>
<td>libXMI.UML.Nodes.UmlParameter</td>
<td>0%</td>
</tr>
<tr>
<td>libXMI.UML.Nodes.UmlPin</td>
<td>16.6%</td>
</tr>
<tr>
<td>libXMI.UML.Nodes.UmlPrimitiveType</td>
<td>0%</td>
</tr>
<tr>
<td>libXMI.UML.Nodes.UmlSendSignalAction</td>
<td>53.5%</td>
</tr>
<tr>
<td>libXMI.UML.Nodes.UmlStructuredActivityNode</td>
<td>46.8%</td>
</tr>
<tr>
<td>libXMI.UML.Stereotypes.CForLoop</td>
<td>24.4%</td>
</tr>
<tr>
<td>libXMI.UML.Stereotypes.CLocalVariable</td>
<td>31%</td>
</tr>
<tr>
<td>libXMI.UML.Stereotypes.ConcurrencyMainFlow</td>
<td>0%</td>
</tr>
<tr>
<td>libXMI.UML.Stereotypes.NonInterruptableRegion</td>
<td>28%</td>
</tr>
<tr>
<td>libXMI.UML.Stereotypes.RcvEvent</td>
<td>20.4%</td>
</tr>
<tr>
<td>libXMI.UML.Stereotypes.SndSig</td>
<td>21.8%</td>
</tr>
<tr>
<td>libXMI.UML.Stereotypes.SwitchDefaultFlow</td>
<td>0%</td>
</tr>
<tr>
<td>libXMI.UML.Stereotypes.SwitchStereotype</td>
<td>21.7%</td>
</tr>
<tr>
<td>libXMI.UML.Structure.UmlActivity</td>
<td>0%</td>
</tr>
<tr>
<td>libXMI.UML.Structure.UmlAssociation</td>
<td>0%</td>
</tr>
<tr>
<td>libXMI.UML.Structure.UmlBehavior</td>
<td>14.2%</td>
</tr>
<tr>
<td>libXMI.UML.Structure.UmlConnector</td>
<td>0%</td>
</tr>
<tr>
<td>libXMI.UML.Structure.UmlConnectorEnd</td>
<td>0%</td>
</tr>
<tr>
<td>libXMI.UML.Structure.UmlPort</td>
<td>0%</td>
</tr>
<tr>
<td>libXMI.UML.Structure.UmlProperty</td>
<td>0%</td>
</tr>
<tr>
<td>libXMI.UML.UmlClass</td>
<td>0%</td>
</tr>
<tr>
<td>libXMI.UML.UmlContext</td>
<td>0%</td>
</tr>
<tr>
<td>libXMI.UML.UmlLiteral</td>
<td>0%</td>
</tr>
<tr>
<td>libXMI.UML.UmlModel</td>
<td>0%</td>
</tr>
<tr>
<td>libXMI.UML.UmlNamedObject</td>
<td>28.5%</td>
</tr>
<tr>
<td>libXMI.UML.UmlObject</td>
<td>44.7%</td>
</tr>
<tr>
<td>libXMI.UML.UmlOperation</td>
<td>7.6%</td>
</tr>
</tbody>
</table>
D.1. Assemblies

libXMI.UML.UmlPackage 0%
libXMI.XMI.Mapping.XmiNodeAttributes 0%
libXMI.XMI.Mapping.XmiNodeTypes 0%
libXMI.XMI.PositionXmlAttribute 0%
libXMI.XMI.PositionXmlDocument 0%
libXMI.XMI.PositionXmlElement 0%
libXMI.XMI.XmiDocument
libXMI.XMI.XmiNode 0%
libXMI.XMI.XmiParser 0%
libXMI.XMI.XmiTranslator 0%
libXMI.XMI.XmiUtil 0%

**UnitTestActivityGen** **98.8%**
UnitTestActivityGen.ActualOutput 57.1%
UnitTestActivityGen.ExpectedOutput 100%
UnitTestActivityGen.UnitTestActActionNodes 100%
UnitTestActivityGen.UnitTestBranch 100%
UnitTestActivityGen.UnitTestCombined 100%
UnitTestActivityGen.UnitTestEvaluation 91.8%
UnitTestActivityGen.UnitTestLocalVariables 100%
UnitTestActivityGen.UnitTestLoop 100%
UnitTestActivityGen.UnitTestMerge 100%
UnitTestActivityGen.UnitTestSignal 100%
UnitTestActivityGen.UnitTestStructuredNode 100%
D. Test Coverage

D.2. ActivityGen.Algorithm

D.2.1. Summary

Class: ActivityGen.Algorithm
Assembly: ActivityGen
Covered lines: 543
Uncovered lines: 0
Coverable lines: 543
Total lines: 799
Line coverage: 100%
Branch coverage: 100%

D.2.2. Metrics

<table>
<thead>
<tr>
<th>Method</th>
<th>Cyclomatic Complexity</th>
<th>Sequence Coverage</th>
<th>Branch Coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>.ctor(...)</td>
<td>1</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>compute()</td>
<td>4</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>transformStructuredA</td>
<td>9</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>transformStructuredN</td>
<td>3</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>createCLocalVariable</td>
<td>11</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>mergeBranches()</td>
<td>13</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>buildCSwitch(...)</td>
<td>12</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>buildCIf(...)</td>
<td>11</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>mergeLoops()</td>
<td>8</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>buildCLoop(...)</td>
<td>14</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>mergeCNodes()</td>
<td>4</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Visit(...)</td>
<td>1</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>setupTimer(...)</td>
<td>4</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>transformAction2C()</td>
<td>2</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>DataStoreOrder()</td>
<td>2</td>
<td>100</td>
<td>100</td>
</tr>
</tbody>
</table>
E. Integration Tests

E.1. Gcd

Figure E.1.: Gcd: Model main function

Figure E.2.: Gcd: Internal graph (control flow edges) main function
Figure E.3.: Gcd: Model gcd function

Figure E.4.: Gcd: Internal graph (control flow edges) gcd function
E.2. Calculator

```c
errInit = OAL_Init();
calc = OAL_TaskSpawn(2, 11, 128, null, calculator);
opSign = OAL_TaskSpawn(1, 10, 128, &calc, operator);
OSStart();
```

**Figure E.5.: Calculator: Model main function**

**Figure E.6.: Calculator: Internal graph (control flow edges) main function**
E. Integration Tests

Figure E.7.: Calculator: Model calculator function

Figure E.8.: Calculator: Internal graph (control flow edges) calculator function
Figure E.9.: Calculator: Model operator function

Figure E.10.: Calculator: Internal graph (control flow edges) operator function
E. Integration Tests

E.3. Structured Nodes

Figure E.11.: Iteration 3: Model structureNode function

Figure E.12.: Iteration 3: Internal graph (control flow edges) structureNode function
F. Static Code Analysis

F.1. Integration Test 1

```c
/* Had to be added manually. */
#include <stdio.h>

int ggt (int a, int b)
{
    if (a == 0)
    {
        return b;
    }
    else
    {
        while (b != 0)
        {
            if (a > b)
            {
                a = a - b;
            }
            else
            {
                b = b - a;
            }
        }
        return a;
    }
}

int main()
{
    int a = 4;
    int b = 2;
    int ggt = ggt(a, b);
    printf("Der ggt ist %i\n", ggt);
    return 0;
}
```

Listing F.1: source code for the Greatest Common Divisor.
Listing F.2: source code for the Greatest Common Divisor with a single point of exit.
<table>
<thead>
<tr>
<th>Result</th>
<th>Entity</th>
<th>Line</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.6 The Standard Library input/output functions shall not be used</td>
<td>printf</td>
<td>33</td>
<td>8</td>
</tr>
<tr>
<td>stdio.h input/output function printf used in file ggt_neu.c</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15.5 A function should have a single point of exit at the end.</td>
<td>ggt</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Multiple exit points from function</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.3 An identifier declared in an inner scope shall not hide an identifier declared in an outer scope</td>
<td>ggt</td>
<td>31</td>
<td>12</td>
</tr>
<tr>
<td>Identifier &quot;ggt&quot; possibly hiding outer definition from line 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5.8 Identifiers that define objects or functions with external linkage shall be unique</td>
<td>ggt</td>
<td>31</td>
<td>12</td>
</tr>
<tr>
<td>Non unique external identifier ggt conflicts with entity ggt in file ggt_neu.c on line 4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.2 Function types shall be in prototype form with named parameters</td>
<td>main</td>
<td>25</td>
<td>4</td>
</tr>
<tr>
<td>Keyword 'void' not used to denote empty parameter list for function main</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.4 A compatible declaration shall be visible when an object or function with external linkage is defined</td>
<td>ggt</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Object or function ggt, has external linkage but no visible declaration</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.6 (Strict</td>
<td>Special ) An identifier with external linkage shall have exactly one external definition</td>
<td>printf</td>
<td>33</td>
</tr>
<tr>
<td>Identifier printf, does not have a definition</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.7 Functions and objects should not be defined with external linkage if they are referenced in only one translation unit</td>
<td>ggt</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Function or object ggt, has external linkage but is only used in the translation unit built from ggt_neu.c.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8.8 The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage</td>
<td>ggt</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>C Function 'ggt' appears to have internal linkage within ggt_neu.c, so static keyword should be used.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table F.1.: Results for iteration 1
void calculator(void *pArg)
{
    int32_t msg = 0;

    int32_t senderTaskId = -1;
    int8_t buffer[10] = { 0 };
    int32_t maxMsgLength = sizeof(buffer);

    int32_t msgId = -1;

    char opSign = ' ';
    int8_t c = 0;

    const int8_t b = 1;
    const int8_t a = 5;

    msg = OAL_MsgRecvBlk(&msgId, buffer, &maxMsgLength, &senderTaskId, 10000);
    if (msgId != -1)
    {
        opSign = buffer[0];
    }
    else
    {
        NOOP;
    }

    switch (opSign)
    {
        case '*':
            c = a * b;
            break;
        case '/':
            c = a / b;
            break;
        case '-':
            c = a - b;
            break;
        case '+':
            c = a + b;
    }
Listing F.3: source code for the calculator.
### F. Static Code Analysis

<table>
<thead>
<tr>
<th>Result</th>
<th>Entity</th>
<th>Line</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.7 There should be no unused parameters in function</td>
<td>Unused unused parameter pArg</td>
<td>3</td>
<td>22</td>
</tr>
<tr>
<td>Unused unused parameter pArg</td>
<td>pArg</td>
<td>51</td>
<td>20</td>
</tr>
<tr>
<td>8.2 Function types shall be in prototype form with named parameters</td>
<td>Improper parameter format for function OAL_MsgRecvBlk; each parameter must have a type and a name</td>
<td>OAL_MsgRecvBlk</td>
<td>21</td>
</tr>
<tr>
<td>Improper parameter format for function printf; each parameter must have a type and a name</td>
<td>printf</td>
<td>47</td>
<td>4</td>
</tr>
<tr>
<td>Keyword &quot;void&quot; not used to denote empty parameter list for function main</td>
<td>main</td>
<td>59</td>
<td>5</td>
</tr>
<tr>
<td>Keyword &quot;void&quot; not used to denote empty parameter list for function OAL_Init</td>
<td>OAL_Init</td>
<td>67</td>
<td>14</td>
</tr>
<tr>
<td>Improper parameter format for function OAL_TaskSpawn; each parameter must have a type and a name</td>
<td>OAL_TaskSpawn</td>
<td>69</td>
<td>11</td>
</tr>
<tr>
<td>Keyword &quot;void&quot; not used to denote empty parameter list for function OSStart</td>
<td>OSStart</td>
<td>72</td>
<td>8</td>
</tr>
<tr>
<td>8.4 A compatible declaration shall be visible when an object or function with external linkage is defined</td>
<td>Object or function calculator, has external linkage but no visible declaration</td>
<td>calculator</td>
<td>3</td>
</tr>
<tr>
<td>Object or function operator, has external linkage but no visible declaration</td>
<td>operator</td>
<td>51</td>
<td>5</td>
</tr>
<tr>
<td>8.6 ( Strict</td>
<td>Special ) An identifier with external linkage shall have exactly one external definition</td>
<td>Identifier OAL_Init, does not have a definition</td>
<td>OAL_Init</td>
</tr>
<tr>
<td>Identifier OAL_MsgRecvBlk, does not have a definition</td>
<td>OAL_MsgRecvBlk</td>
<td>21</td>
<td>10</td>
</tr>
<tr>
<td>Identifier OAL_TaskSpawn, does not have a definition</td>
<td>OAL_TaskSpawn</td>
<td>69</td>
<td>11</td>
</tr>
<tr>
<td>Identifier OSStart, does not have a definition</td>
<td>OSStart</td>
<td>72</td>
<td>8</td>
</tr>
<tr>
<td>Identifier printf, does not have a definition</td>
<td>printf</td>
<td>47</td>
<td>4</td>
</tr>
</tbody>
</table>
8.7 Functions and objects should not be defined with external linkage if they are referenced in only one translation unit

<table>
<thead>
<tr>
<th>Function or object</th>
<th>calculator</th>
<th>3</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>has external linkage but is only used in the translation unit built from calculator.c.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Function or object</th>
<th>operator</th>
<th>51</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>has external linkage but is only used in the translation unit built from calculator.c.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

8.8 The static storage class specifier shall be used in all declarations of objects and functions that have internal linkage

<table>
<thead>
<tr>
<th>C Function &quot;calculator&quot; appears to have internal linkage within calculator.c, so static keyword should be used.</th>
<th>calculator</th>
<th>3</th>
<th>5</th>
</tr>
</thead>
</table>

| C Function "operator" appears to have internal linkage within calculator.c, so static keyword should be used. | operator   | 51 | 5 |

Table F.2.: Results for iteration 2
F.3. Integration Test 3

```c
void structuredNode()
{
    /* this is a body */
    interruptReg = OAL_IntLock();
    /* This is a body in a non interruptable region */
    OAL_IntUnlock(interruptReg);

    regVar = OAL_IntLock();
    for (i = 1; i < 10; i++)
    {
        /* This is a body in a loop */
    }
    OAL_IntUnlock(regVar);
}
```

Listing F.4: source code for the structured nodes.
### Table F.3.: Results for iteration 3

<table>
<thead>
<tr>
<th>Result</th>
<th>Entity</th>
<th>Line</th>
<th>Column</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.2 Function types shall be in prototype form with named parameters</td>
<td>structuredNode</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>Keyword &quot;void&quot; not used to denote empty parameter list for function</td>
<td>structuredNode</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>OAL_IntLock</td>
<td>OAL_IntLock</td>
<td>5</td>
<td>19</td>
</tr>
<tr>
<td>Improper parameter format for function</td>
<td>OAL_IntUnlock</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>OAL_IntUnlock; each parameter must have a type and a name</td>
<td>OAL_IntUnlock</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>8.4 A compatible declaration shall be visible when an object or function</td>
<td>structuredNode</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>with external linkage is defined</td>
<td>structuredNode</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>8.6 ( Strict</td>
<td>Special ) An identifier with external linkage shall have exactly one</td>
<td>OAL_IntLock</td>
<td>5</td>
</tr>
<tr>
<td>external definition</td>
<td>OAL_IntLock</td>
<td>5</td>
<td>19</td>
</tr>
<tr>
<td>Identifier OAL_IntUnlock, does not have a definition</td>
<td>OAL_IntLock</td>
<td>5</td>
<td>19</td>
</tr>
<tr>
<td>Identifier OAL_IntUnlock, does not have a definition</td>
<td>OAL_IntUnlock</td>
<td>7</td>
<td>4</td>
</tr>
<tr>
<td>8.7 Functions and objects should not be defined with external linkage</td>
<td>structuredNode</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>if they are referenced in only one translation unit</td>
<td>structuredNode</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>C Function 'structuredNode' appears to have internal linkage within</td>
<td>structuredNode</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>resultIteration3.c, so static keyword should be used.</td>
<td>structuredNode</td>
<td>1</td>
<td>5</td>
</tr>
</tbody>
</table>
Bibliography


